eStream client management functions include the following:

- For ASP accounts known to this client:
  - View account information **[LSMGetAspList, LSMGetAspInfo]**
  - Request synchronization of application subscription information [handled automatically at client eStream activation, included here for flexibility] **[LSMUpdateAllSubscriptionStatus]**
- For application subscriptions known to this client:
  - View subscription information [this may engender a synch] **[LSMGetAppList, LSMGetAppInfo]**
  - Install subscribed application [handled automatically at synch time, included here for flexibility] **[LSMInstall]**
  - Uninstall application [handled automatically at synch time, included here for flexibility] **[LSMUninstall]**
  - Upgrade application to latest version [handled automatically, flexibility] **[LSMUpgrade]**
- For eStream client software:
  - Specify automatic activation at login vs manual activation via program/start [it is proposed that we assume the former is always the case] **[CESSetStartup]**
- For eStream client cache [primarily for users who do not allow eStream to right-size their cache]:
  - Display current size and utilization [obtained from registry]
  - Set size: allow eStream to right-size vs. manually increase/decrease size **[ECMSetCacheSize]**
  - Disable/enable prefetching **[EPFPrefetchSet]**
- For eStream client network interface:
  - Display recent eStream effective bandwidth [possibly displayed as hover-over on eStream systray icon] **[CNIGetEffectiveBandwidth]**
  - View/Set proxy IP address [may use info from control panel internet options widget] **[CNIGetProxyIPAddress, CNISetProxyIPAddress]**
- Deactivate eStream **[CESShutdown]**

CUI Interfaces To/From Client Components

Client Component Acronyms

CES: Client EStream Startup
CNI: Client Network Interface
CUI: Client User Interface
ECM: EStream Cache Manager
EPF: EStream PreFetch
LSM: License Subscription Manager

From CUI to CES:

CESSetStartup(IN Boolean ActivateAtLogin)
CESShutdown(VOID)

From CUI to LSM:
LSMGetAspList(OUT NumAsps, OUT AspID[])
LSMGetAspInfo(IN AspID, OUT ASPWebServerName, OUT UserName)
LSMUpdateAllSubscriptionStatus(VOID)
LSMGetAppList(IN AspID, OUT NumApps, OUT AppID[])
LSMGetAppInfo(IN AppID, OUT AppName, OUT VersionName, OUT Message,
OUT AppInstallStatus, OUT AppUpgradeStatus)
LSMInstall(IN AppID, OUT InstallStatus)
LSMUninstall(IN AppID, OUT DeinstallStatus)
LSMUpgrade(IN AppID, OUT UpgradeStatus)

From CUI to ECM:
ECMSetCacheSize(VOID) -- causes ECM to recheck registry for updated cache size

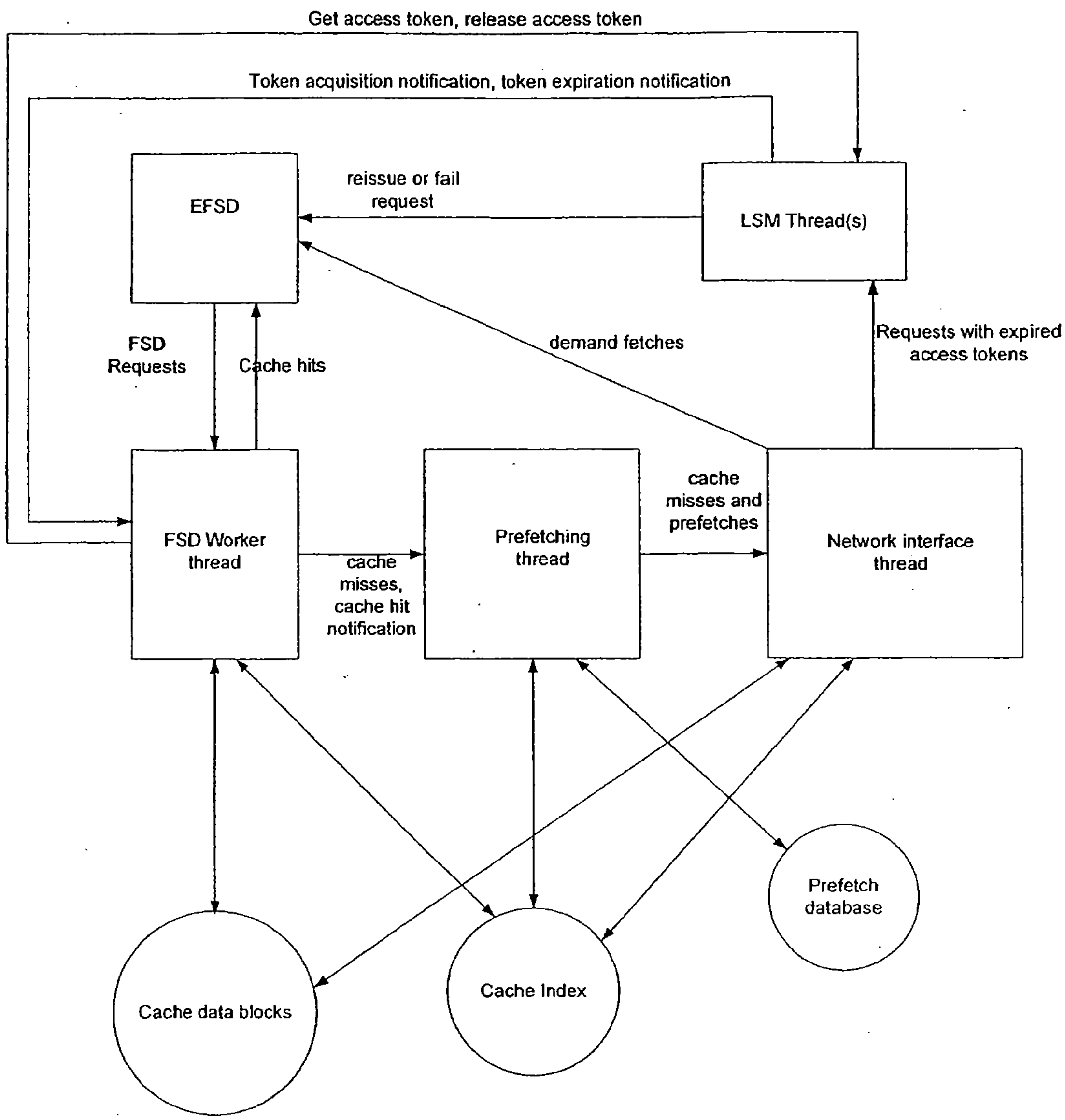
From CUI to EPF:
EPFPrefetchSet(IN OnOff)

From CUI to CNI:
CNIGetProxyIPAddress(OUT ProxyIPAddress)
CNISetProxyIPAddress(IN ProxyIPAddress)
CNIGetEffectiveBandwidth(OUT EstreamRecentEffectiveBandwidth)

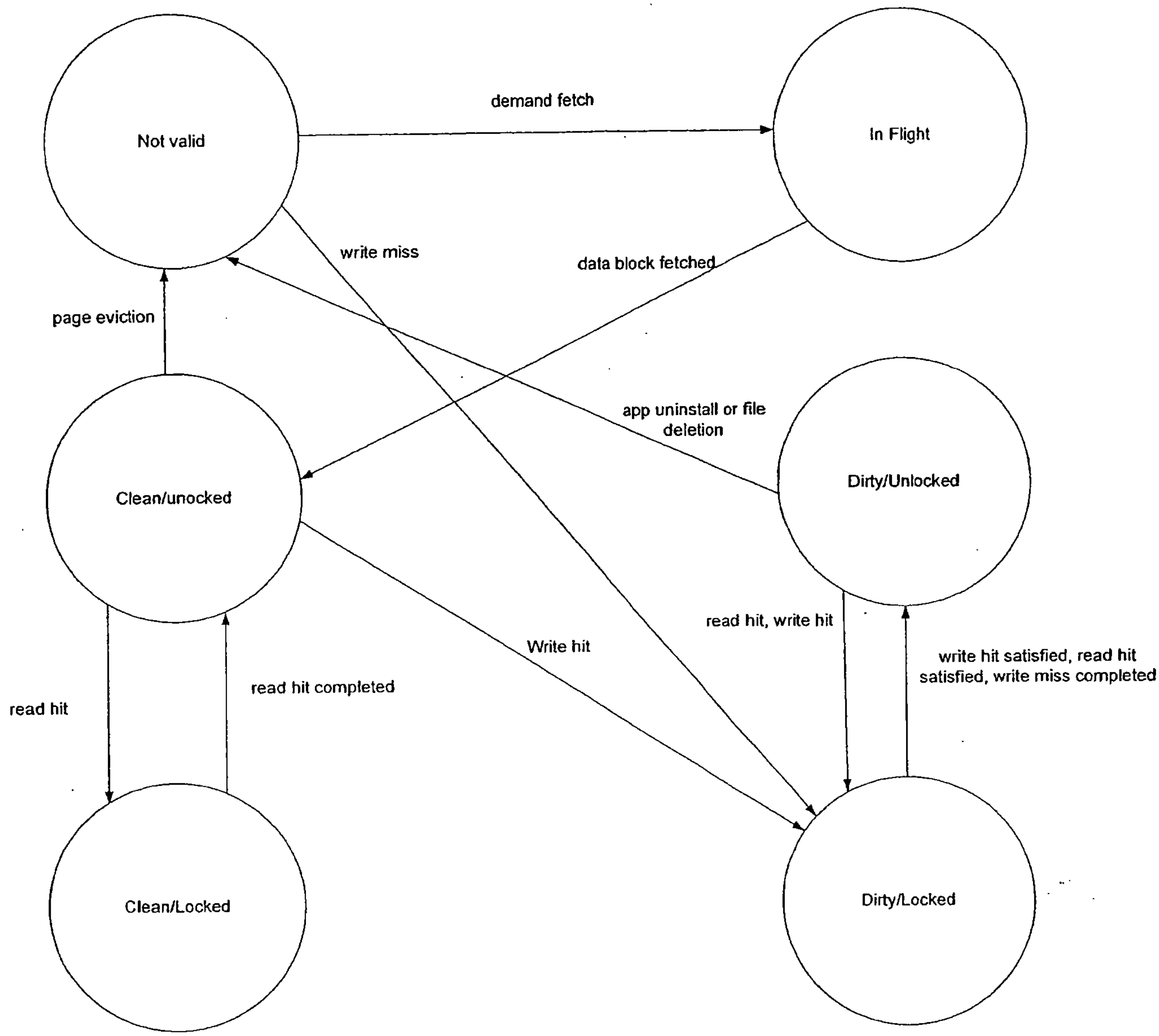
From <various client components> to CUI:
CUIInformUser(IN Message, IN OptionalHelpInfo)
CUIAskUserYesNo(IN Message, IN OptionalHelpInfo, IN CheckAskAgain,
OUT Response, OUT DontAskAgain)
CUIAskUserPassword(IN Message, OUT Response, OUT Retain)

Issues

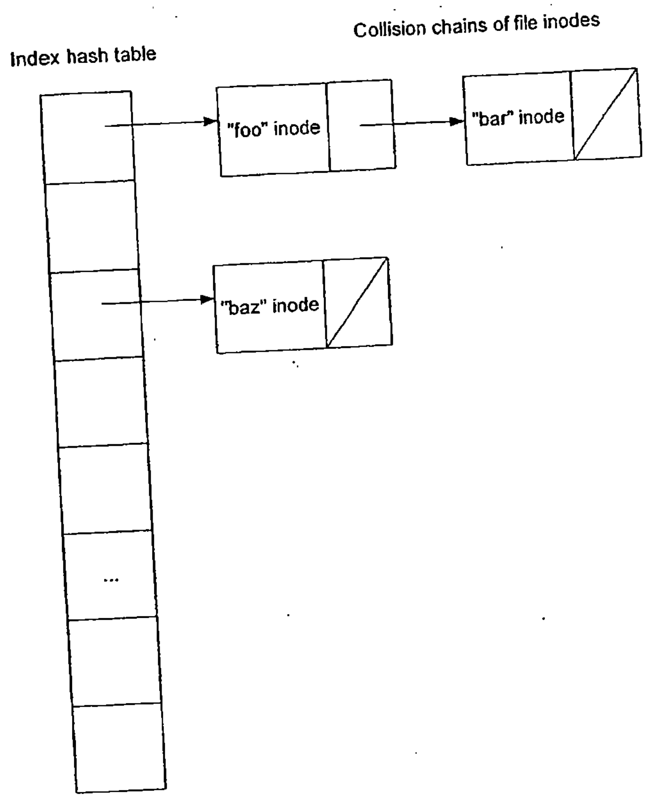
- Resolve general issue of whether eStream client management functions should be included in product at all (opinions on both sides).
- Within client management functions, should prefetch enable/disable be included?
- Should CNI get ProxyIPAddress from control panel internet options widget and should we therefore remove this menu item & interface from CUI?
- Will cache utilization be posted to the registry or available somehow?
- What is the interface to obtain recent effective eStream network bandwidth?

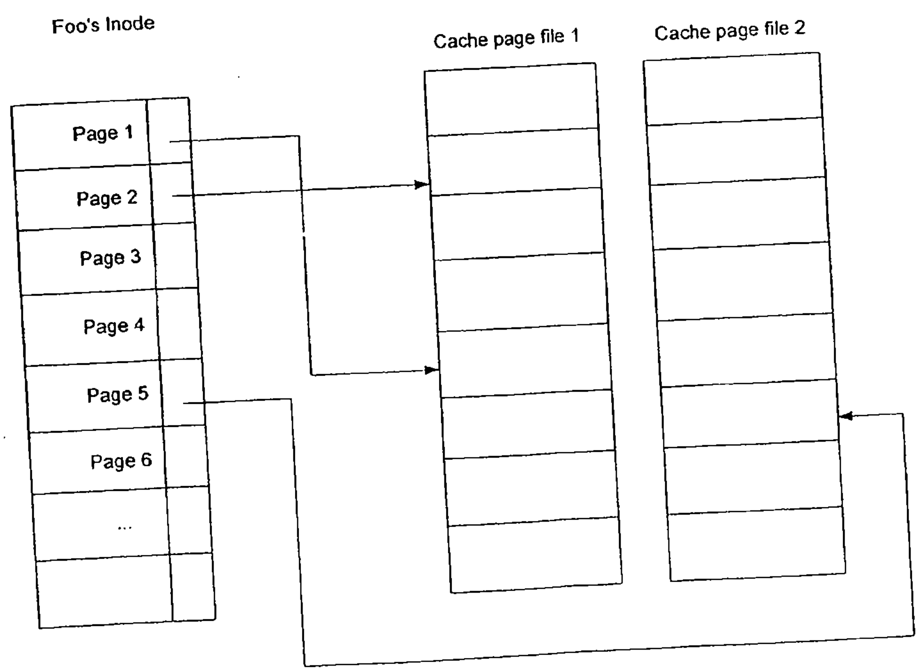


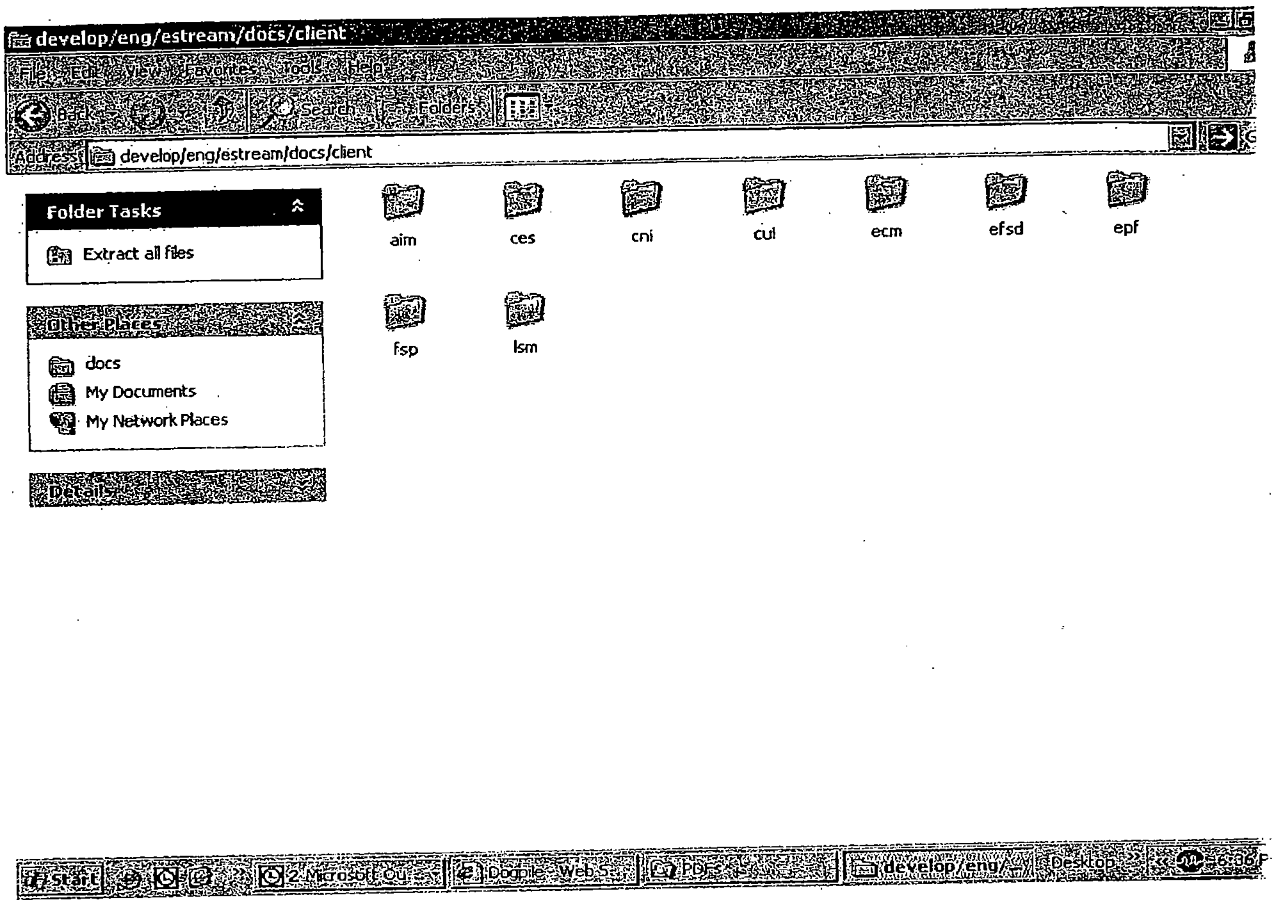
Get access token, release access token
Token acquisition notification, token expiration notification
EFSD
reissue or fail request
LSM Thread(s)
FSD Requests
Cache hits
demand fetches
Requests with expired access tokens
FSD Worker thread
Prefetching thread
cache misses and prefetches
Network interface thread
cache misses, cache hit notification
Prefetch database
Cache data blocks
Cache Index



Not valid
demand fetch
In Flight
write miss
data block fetched
page eviction
app uninstall or file deletion
Clean/unocked
Dirty/Unlocked
read hit, write hit
Write hit
write hit satisfied, read hit satisfied, write miss completed
read hit
read hit completed
Clean/Locked
Dirty/Locked


Collision chains of file inodes
Index hash table
"foo" inode
"bar" inode
"baz" inode
...


Foo's Inode
Cache page file 1
Cache page file 2
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
...


develop/eng/estream/docs/client
Address develop/eng/estream/docs/client
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm

eStream 1.0 Cache Manager Low Level Design

Version 1.4

[redacted]

Omnishift Technologies, Inc.
Company Confidential

Functionality

The eStream cache manager implements much of the client-side functionality for handling the eStream file system. The cache manager handles all file system requests made by the operating system by reading information from the cache or by passing the requests along to the profiling and prefetching component to fetch missing data from the network.

The cache manager will initially be implemented in user space, but it may be useful to migrate it to the kernel for improved performance. In user space, it will be part of the eStream client process. In the kernel, it will probably be a device driver distinct from the eStream file system driver.

The cache manager manages the on-disk cache of file system data, and the in-memory data structures for managing this cache. It does not manage prefetching of data from the server; that is the role of the eStream Profiling and Fetching (EPF) component. A separate networking component handles the network traffic. This component will also be described separately.

Since there is no overall discussion of the client architecture at a more detailed level than the high level design, this document will cover that as well.

Multiple cache page files will be supported. Each cache page file may be up to 2 GB in size. Different cache files may reside on different or the same logical disk (i.e. Windows drive letter.)

Data type definitions

An application ID uniquely identifies an eStream application. Just what constitutes "one" eStream application is not entirely defined, but different "builds" of the "same" app will be considered different eStream applications. For example, the Chinese-language version of Office is a different eStream application than the English-language version.

```
typedef uint128 ApplicationID;
```

The eStream page number is the data type used to describe a page number within a particular file. Note that this is a page offset, not a byte offset. For eStream 1.0, the cache manager will only support 2 GB cache files.

```
typedef uint32 EStreamPageNumber;
```

The fileId is used to uniquely identify a file within the universe of all eStream files across all eStream applications.

```
typedef struct {
     ApplicationID App,
     int32 File
} fileId;
```

The eStream page size is the fundamental size for eStream requests. This size is in bytes.

```
#define ESTREAM_PAGE_SIZE 4096
```

The eStream file system uses the file time format of the Windows operating system. If the client runs on a system with a different native time format, the client software will be responsible for translating between the native format and the eStream format. The Windows data format is a 64-bit counter of the number of 100-nanosecond periods since January 1, 1601.

EStream metadata is the file information supported by the eStream file system. This metadata is independent of the client or server operating system.

```
typedef struct
{
     uint64 CreationTime;
     uint64 AccessTime;
     uint32 FileSize;
     uint32 FileSystemAttributes;
     uint32 EStreamAttributes;
} Metadata;
```

The eStream inode contains the layout of a file in the cache. Each inode has the following structure:

```
typedef struct
{
     FileId Id; /* ID of this file; search parent for
name*/
     Metadata Metadata;
     FileID Parent; /* parent directory's file id */
     uint32 NumPages;
     PageInfo *Pages;
} EStreamInode;
```

The PageInfo array is variable sized. There is one entry in the pages array for each page in the file (not for each page cached, since we need to know whether the pages are present or not...) Note that the inode is only used in the "robust" implementation.

```
typedef struct
{
	EStreamPageNumber CachePageNumber;
	PageStatus Status;
	unsigned char Priority;
	PageChecksum Checksum;
} PageInfo;
```

The page number doesn't require the 32 bits, since pages are 4096 bytes long. The extra bits will be used to encode which cache file this page resides in. The priority field is a number representing this page's priority for being kicked out of the cache. How exactly this field is used hasn't yet been determined. The checksum is a (fast) page checksum that can be used to validate the contents of this page. Note that it will be useful to have a slower, more effective checksum for development and a faster (but less thorough) checksum for deployment.

The page status is an enumeration for the page's locking status (these are described in more detail later:

```
typedef enum
{
	PS_INVALID,
	PS_CLEAN_UNLOCKED,
	PS_CLEAN_LOCKED,
	PS_DIRTY_UNLOCKED,
	PS_DIRTY_LOCKED,
	PS_IN_FLIGHT
} PageStatus;
```

Note that this describes the layout of the tables in memory; how these data structures are represented on disk is described later.

The EFSD file handle is a small integer passed between the EFSD and the ECM. This is used opaquely by the EFSD and is used as an index into an open file table by the ECM.

```
typedef uint32 EFSDFileHandle;
```

The ECM request type specifies the request type to the rest of the system. Note that some "requests" are used to inform the prefetcher about the events handled solely by the ECM, and do not actually request that any particular action be taken by the prefetcher.

```
typedef enum
{
	ERT_READ,
	ERT_WRITE,
	ERT_READ_HIT,
```

```
    ERT_WRITE_HIT
} ECMRequestType;
```

The ECM request is a request descriptor that is used in various lists within the cache manager. These lists are doubly-linked, circular lists.

```
typedef struct _ECMRequest
{
    uint32 RequestID; /* same as EFSD request id */
    ECMRequestType RequestType;
    union {} Parameters; /* union of all parameters*/
    struct _ECMRequest *next;
    struct _ECMRequest *prev;
} ECMRequest;
```

The cache manager must maintain an array of files that have currently been opened by the EFSD. This array will be statically allocated. This will put a limit on the number of files that may be opened concurrently on the eStream file system. The elements of the array are the following:

```
typedef struct
{
    uint32 Valid;
    fileId File;
    HANDLE OpenFile; /* for simple implementation */
    eStreamInode *Inode; /* for robust implementation */
} OpenFileInfo;
```

The cache manager maintains a hash table containing information about each application that currently has open files. The hash table is indexed by app ID, and contains the following active app information records:

```
typedef struct
{
    AppID App; /* identity of this app */
    uint32 OpenFiles; /* # of open files */
    uint32 HaveAccessToken; /* boolean */
} ActiveAppInfo;
```

The ECM will use this table to quickly determine whether it should continue processing a request it gets from the EFSD, or if the request should be passed to the LSM to ensure that an access token is available. See the section below on ECM-LSM interaction for more details.

The LSM uses the access token state to specify a state for an access token. Right now, we only plan to support valid and invalid, but it may be interesting in the future to allow already opened files to be read, but no new files to be opened.

```
typedef enum
{
     ATS_INVALID,
     ATS_VALID,
     ATS_VALID_NO_OPEN
} AppTokenState;
```

Interface definitions

The ECM exports the following interfaces for operating on the cache. They may be called by the cache manager, prefetcher, or networking component. (Not all components are expected to call all interfaces; see each interface description for more details.)

Note that the cache interfaces are defined at a very high level as the actions that may be performed on the cache by the components, such as enqueuing a new request. They have been defined this way so that these intrinsic operations can be implemented correctly once and limit the possibility that an individual component will not perform proper actions.

ECMReservePage

eStreamStatus ECMReservePage(
    IN fileId *File,*
    IN EStreamPageNumber *Page,*
    IN ECMRequest **Request*
);

ECMReservePage reserves a page in the cache for a request. This interface is called by the prefetching component, and will send a request to the network component. Logically, this interface reserves an empty cache page for this request (if one is available), puts this request on the "in flight" queue, and calls on the network to request the page (unless it is already in flight.)

ECMIsPageInCache

eStreamStatus ECMIsPageInCache(
    IN fileId *File,*
    IN EStreamPageNumber *Page*
);

ECMIsPageInCache returns TRUE if the specified block is in the cache, and FALSE otherwise. It is used by the EPF to determine if it should prefetch a block; normally, the EPF would choose not to prefetch something that is already in the cache. Note that it would be a good idea for the prefetcher to adjust the priority of a page that it thinks it wants to prefetch, so that they are less likely to be evicted from the cache before they are needed.

ECMDeplanePage

eStreamStatus ECMDeplanePage(

IN fileId *File,*
IN EStreamPageNumber *Page,*
IN char *Buffer[ESTREAM_PAGE_SIZE]*
);
ECMDeplanePage performs all the necessary actions for writing a page coming off the network into the cache and back to the EFSD. This consists of copying the page into the cache, remove all pending requests for this page from the in flight list, marking the page as clean/unlocked, and returning the page to the EFSD for each in flight request.

ECMReadPage

eStreamStatus ECMReadPage(
IN fileId *File,*
IN EStreamPageNumber *Page,*
IN ECMRequest **Request*
);
ECMReadPage performs all the necessary actions for attempting a page read from the cache. The cache is checked to see if it contains the page; if so, the page is copied to the buffer, the EPF is notified of the hit, and appropriate status is returned. Otherwise, this page is put on the queue for requests pending to the prefetching component, and appropriate status is returned.

ECMWritePage

eStreamStatus ECMWritePage(
IN fileId *File,*
IN EStreamPageNumber *Page,*
IN ECMRequest **Request*
);
ECMWritePage performs all the necessary actions for attempting to write a page in the cache. Note that this could be somewhat more complex than a read, because a partial write to a page might necessitate reading the page from the server before writing the partial page to the cache.

The following interfaces are the abstract interfaces that the ECM will use to communicate with the EFSD. Hiding the EFSD's raw DeviceIoControls behind these interfaces will help make porting the ECM into the kernel easier, should we decide to do that.

ECMSetTokenState

eStreamStatus ECMSetTokenState(
IN AppId *App,*
IN AppTokenState *State*
);
ECMSetTokenState is called by the LSM to indicate to the ECM that a token has become available or has expired. The main effect of this interface is to update the state of the specified application in the active app table. See the ECM-LSM interaction below for more details.

ECMGetCacheInfo

eStreamStatus ECMGetCacheInfo(
OUT UNICODE_STRING *Location,*
OUT uint32 **Current*Size,*
OUT uint32 **MaximumSize*
);

ECMGetCacheInfo is called by the client user interface to find out where the ECM cache is located and its current and maximum size. Location is an absolute path name of the cache file.

ECMSetCacheInfo

eStreamStatus ECMSetCacheInfo(
IN UNICODE_STRING *Location,*
IN uint32 *MaximumSize*
);

ECMSetCacheInfo is called by the user interface when a new cache location or size has been requested. Note that the cache manager may only begin using the new cache information after a restart of the client software (which may only occur on client machine reboot.) The client UI will call this interface when it wants to make a change; the ECM is responsible for actually resizing the cache and making any changes necessary to persistent storage (i.e. the registry).

EFSDGetRequest

eStreamStatus EFSDGetRequest(
OUT EStreamRequest ***Request*
);

EFSDGetRequest reads the next request from the EFSD, including any parameters that need to be passed. This may involve one or more DeviceIoControl calls to the EFSD. **EFSDGetNextRequest** is responsible for allocating memory for this request, and an **EFSDCompleteRequest** call will be repsonsible for deallocating the memory.

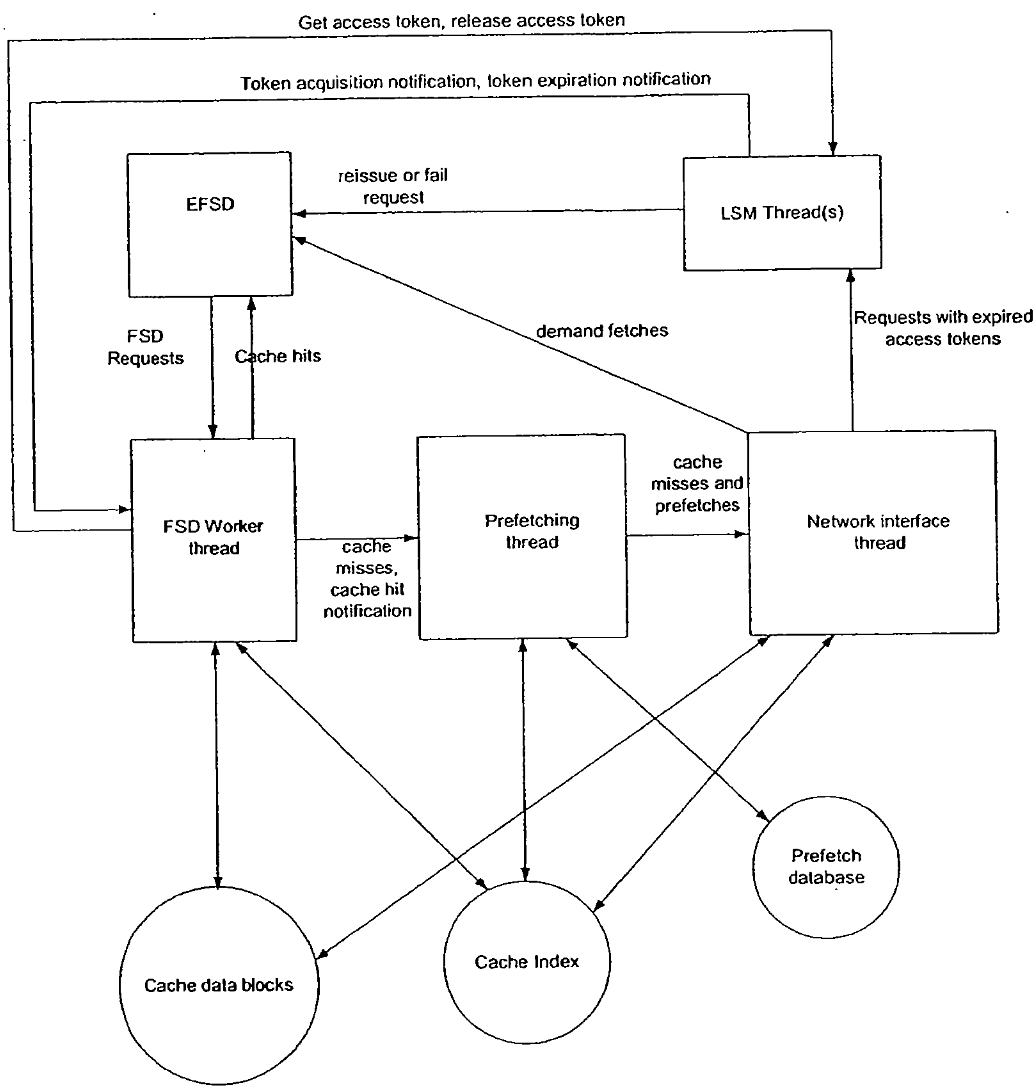
EFSDCompleteRequest

eStreamStatus EFSDCompleteRequest(
IN EStreamRequest **Request,*
IN ECMErrorCode *Status*
);

EFSDCompleteRequest will be called for each request that is received by the ECM via EFSDGetRequest. *status* indicates the completion status for this request, and may indicate success, a retry, or a particular failure condition. Non-persistent errors will be handled by the ECM internally or by requesting a retry of a particular request. Errors reported to the EFSD will be propagated up the file system stack.

Overall Client Architecture

The eStream client will have various types of threads in order to perform its work. The basic architecture is illustrated by the following diagram.



The FSD worker thread will pull requests from the FSD. It will return data for requests that can be satisfied immediately. Any request that requires information that is not currently in the cache will be put on a queue for the prefetching thread to handle.

The profiler will receive all cache misses from the FSD worker thread. Using its own data structures (which may include information about recent cache misses in addition to information about general prefetch patterns), it will decide which blocks it should prefetch. Demand fetch and prefetch requests are sent to the network component. The

only way demand fetches and prefetches are treated differently by the network component is that demand fetches are sent to the EFSD while prefetches are not.

The network thread will manage open connections to app servers and retry requests that time out. When data comes back from the network, the network thread will copy the returned buffer into the cache and to the FSD, if the request was a demand miss.

The cache manager consists of the EFSD worker thread and the APIs to access the cache index, the data blocks, and various queues used by threads in the client.

Not shown on the diagram is an error thread. This thread is responsible for calling the client UI module indicating appropriate error messages and waiting for the user's input. When any component decides that it has an error condition that requires user input, it calls **ECMReportError** with the request and an appropriate error condition, which will be enqueued for the error thread to handle. For example, when the network interface times out reading a page from an application server enough times, it will call **ECMReportError**. When the error thread gets to this request in the queue, it will ask the user if he wants to wait until the app server is available or allow the application to terminate.

ECM-LSM Interaction

The ECM-LSM interaction is a relatively simple one. The LSM notifies the ECM when it first receives an access token and when its access token expires. It does this via the **ECMSetTokenState** interface. The ECM keeps track of each application that has had files open, and whether or not we have an access token for each of these apps.

App ID	# of open files	Have access token?

Note that the LSM need not notify the ECM of mundane events like renewals as long as some token is valid. Also, the ECM does not keep track of the token itself, just whether or not we have a valid one. An additional nicety of this approach is that we could allow the ECM to satisfy requests out of the cache as if we have an access token, without actually having one.

When it receives a request, the ECM checks its table to determine if an access token is available. If it is, it handles the request as normal. If not, it asks the LSM to acquire an access token via **LSMGetAccessToken**. The LSM may return that it has a token, in which case the ECM will continue to process the request, or the LSM may say it doesn't have a token, in which case the LSM takes ownership of the request and will reissue the request when the access token is available.

When the number of open files drops from 1 to 0, the ECM will mark the token as invalid in its table and call **LSMReleaseToken**. The LSM may choose not to renew access tokens that have been released.

Component design

Two cache organizations will be presented. One is suitable for a quick implementation but doesn't lend itself particularly well to high performance or easy manageability; the other will be more difficult to implement but should provide better performance. I will first describe some data structures that are shared by both designs, then go into the specifics of each design.

Common Data Structures and Algorithms

Certain request lists are common to both cache organizations. One is a queue between the FSD worker thread and the prefetching thread for demand fetches that have not yet been seen by the prefetcher. The other is a list of all requests for pages that are "in flight." Requests from the in flight list are removed when they have been satisfied. The in flight list is unsorted and searched whenever a request comes back for requests that match the returned page. If the performance of this data structure becomes an issue, we will change its organization for faster lookup.

Both request lists use the request data structure described above.

The ECM will maintain an array of files currently opened by the EFSD. On file opens, an empty location in this table will be allocated for the newly opened file, and the index to that entry returned as the file handle. (Note that the way the interface between the ECM and the EFSD is defined, it is an error to open an already opened file. The cache manager will have to detect such cases and report an error, but it will not keep a reference count of the number of opens on each file.) This mechanism will allow the ECM to keep track of the volumes that currently have opened files as well as abstracting the client/server file ids away from the kernel driver. (This might allow us to update the client/server protocol without rewriting the EFSD.)

Easier Implementation

The cache will be implemented as a directory tree on the user's hard drive that parallels the eStream file system. Each file will contain a header and an array of status bytes in addition to the data blocks that the file contains. The array of status bytes has one byte for each page in the file. Each byte indicates the current status of that page in the file. (Pages have several different states, so a simple bit per page is not sufficient.) Each file will thus look like

Header
Page Status Bytes
File contents page 0

File contents page 1
...

The header is defined as:

```
typedef struct
{
uint32 magicCookie;
uint32 headerLength; /* Length of this header, in bytes */
fileId fileid; /* for sanity checking */
uint32 length; /* Length of the file, in bytes */
uint32 firstPage; /* Offset to the first page in the file
*/
Metadata metadata;
} ECMCacheFileHeader;
```

The page status bytes begin immediately following the header, and this area is padded with zeros to a page boundary. The first page of the file's contents (and thus each following page of file contents) will therefore begin on a page boundary.

Note that one issue with this design is that files that approach the file size limit of the underlying file system cannot be represented, due to the overhead with the header and bitmap. If this design is used solely for early engineering efforts, then this limitation is acceptable. If we have to work around this limitation, one way to do it is to make the headers and page status bytes reside in a separate file or files.

Directory contents would reside in server format in a file named "Directory" inside of the directory whose contents they represent (with the addition of the header and status bytes as described above for ordinary files). For example, z:\Program Files\Microsoft Office would reside in c:\Cache\Program Files\Microsoft Office\Directory. This has the drawback of creating special file names that can't be used by files in the eStream volume, but again, for an early engineering implementation, this is an acceptable limitation.

Another issue with deploying this implementation is that it is trivial to reverse-engineer this file format and copy files directly from the cache.

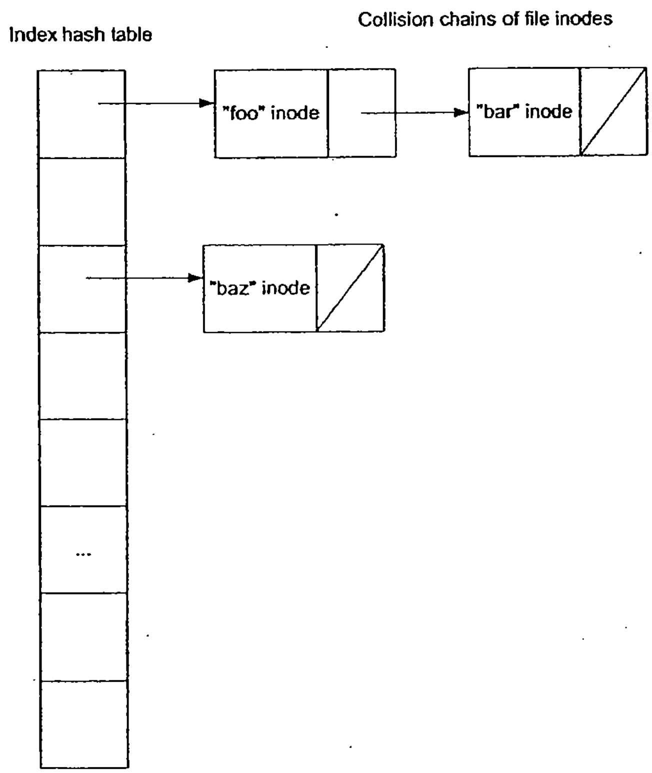
Robust Implementation

The cache will be organized into an index file and one or more cache data files. Multiple data files may be necessary as we may wish to allow the cache to grow larger than the 2 GB file size limit (for some native file systems) or to span multiple drive letters on the client. The data files will only contain pages of file content. These pages will be aligned on page boundaries. The index file contains all the information needed to locate file pages, and is contained in a separate file for simplicity.

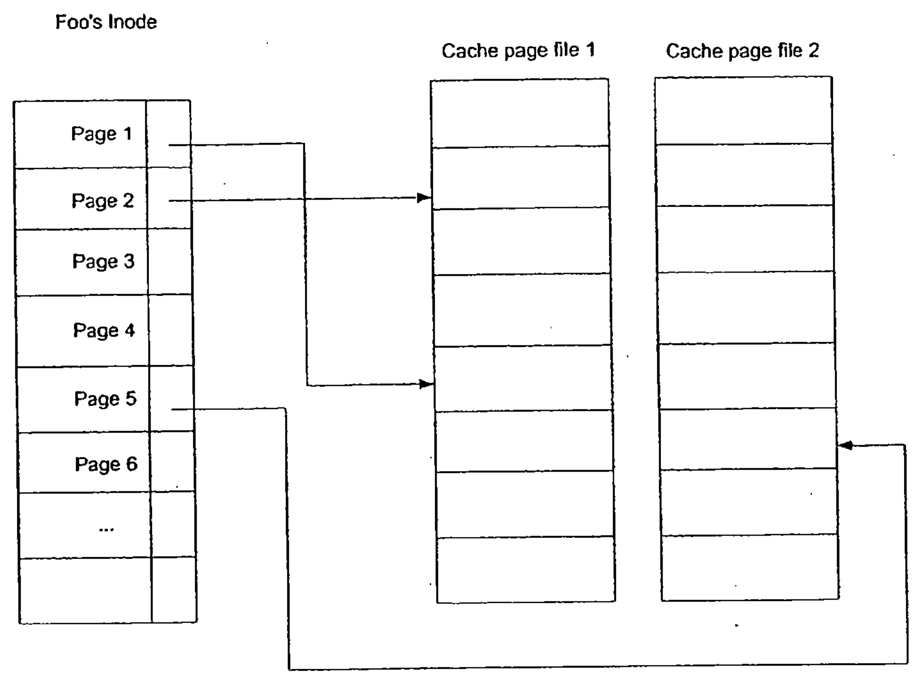
Page and index files must reside on a local disk (rather than a network disk) and cannot be shared by multiple clients.

Each file with any pages currently resident in cache will have a data structure containing information about that file, including its file id, the file id of the directory containing it, the file's metadata, and the map for finding the file's data blocks. This data structure is very similar to the inode of a traditional file system, and will be referred to as the eStream inode. A naive implementation of the inode is described above; no doubt, we will want to reorganize this data structure for more compact representation and better performance. Note that one requirement of the inode is that it contain a status field for each page in the file. One character is sufficient for this status; whether or not we can make do with fewer than 8 bits is an open question.

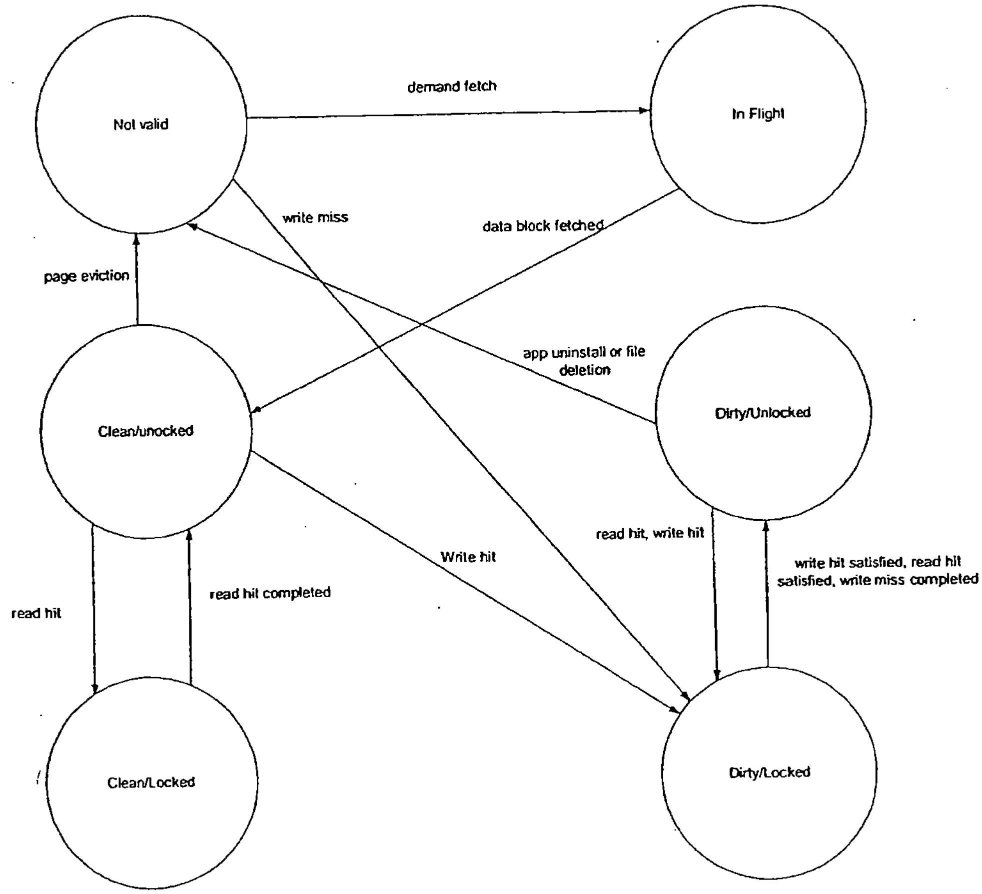
A hash table will be used to map file IDs to file inodes.



The inode contains pointers to each block's location in one or more cache page files:



To prevent race conditions, a single lock controls access to both the hash index and the linked list of requests that are pending network access. Individual pages in the cache may be locked for read or write access. Since each page's status is in the index, the index must be locked order to lock a page for reading or writing. The page states are controlled by the following state machine:



The dirty/clean distinction is between those pages that we have written locally (and thus cannot evict from the cache) and those pages that we haven't written (and thus can be refetched from the server).

A page would be locked while it was being read or written for copying to the file system driver. The operation may thus proceed with the index unlocked, without the possibility of page eviction while a copy is still in progress. The FSD worker thread is the only thread that reads or writes pages from the cache, so it's the only thread that can lock or unlock these pages. The in flight state is only for pages that are currently being fetched, either as a demand fetch or as a prefetch. The prefetching thread is the only thread that will put pages into this state, and only the networking thread will move pages from in flight to unlocked.

A list will be maintained of all "in-flight" requests. A single lock will control access to both this list and the cache index, so there are no race conditions between items being put on this list and data coming off the network. When the FSD worker thread gets a request, it acquires the index lock and looks at the status of the page. If the page is clean or dirty but unlocked, it will lock the page and copy it to the FSD. If the page is invalid, then this is a demand fetch, and the request is forwarded to the prefetcher. If the page is marked in

flight, then this is either a second request for an outstanding demand fetch, or it is a request for an in flight page. Either way, while this thread still holds the index lock, this request will be inserted into the list of in-flight requests. Race conditions might occur because the FSD might make multiple demand reads of the same page, or it may make a demand read to a page that is already in flight due to a prefetch.

Reading requested pages off the network and writing them to the cache (and to the file system driver, if necessary), are where this race condition comes up. We need to ensure that a request for a page that has arrived does not end up in the list of "in flight" requests. The solution is the following: When a data page comes back from the server, the networking component acquires the index lock to find the cache location of this incoming page. If the page is not marked in flight in the cache, this is a bug. (Of course, this is a relatively benign bug, and the NW component could just ignore the page.) The networking thread leaves the page as marked in flight, however, and unlocks the index. It writes the incoming page into the proper location, but it saves the in-memory copy of the page. It then reacquires the index lock, marks the page as clean/unlocked (since it's now in its final location in the cache), removes each request in the in-flight list for this page, then releases the lock. (Any further requests for the same page will find the page clean/unlocked, so the FSD worker thread will be able to satisfy these requests directly.) The networking component then proceeds to satisfy all of the requests it pulled off the in-flight list by using the copy of the page that it saved in memory. This way, it doesn't have to lock the index the entire time it is sending completed requests to the FSD.

Each of these complex scenarios is captured in the cache file's API's. As long as these are implemented correctly, other components don't need to worry about the exact sequence of operations that needs to occur.

Free Space Management

Free pages will be maintained as a free list in memory and as a bitmap on disk. The free list will be built from the bitmap on eStream client software startup. Access to the free list will be controlled by the same lock controlling access to the index.

Evicting Cache Pages

Individual cache pages may be evicted. There is an 8-bit field in the index for each page's importance. Initially, we will implement a random page replacement policy. Later, we will use this page importance field in an unspecified way to replace pages in such a way as to maximize interactive user performance and minimize application server load. Only clean/unlocked pages may be evicted. Pages that are evicted will eventually be put on the free list. Page eviction will only happen at "garbage collection" time. See "crash resilience and garbage collection," below.

Handling Cache Size

Growing the cache should not be an issue. The cache manipulation routines must know the overall size of the cache, in pages. Increasing the size of the cache on the fly should be a relatively straightforward process, as we merely need to lengthen the cache file(s) and add the new pages to the free page list.

Unfortunately, shrinking the cache is a much more difficult operation, since it potentially involves moving around pages that might currently be in use for paging operations or be in flight from the network. Changing the cache around at runtime is both difficult to implement correctly and a performance problem. The current plan is to support shrinking the cache only at eStream client software startup. The maximum allowed size of the cache will be stored in the Registry. On eStream client software startup, the current size of the cache will be compared against the allowed size specified in the registry; if it is larger than the maximum size specified in the registry, then the size of the cache will be reduced by evicting files and compacting the freed space. A request by the user to reduce the size of the cache will take effect the next time the client software starts.

Note that files that the user writes to the z: drive are not considered candidates for eviction (unless the file is explicitly deleted.) This means that the user's on-disk cache may in fact grow to be larger than the limit they specify.

Also note that at least one free page (not used by user-written files) is required for the file system to make forward progress. We also may want to require some minimal amount of cache before eStream will even run. Thus the maximum cache size specified by the user should be considered a "soft limit." There would be a "hard" minimum amount of space equal to the number of pages required to store the files written by the user on the z: drive plus a small amount of cache we designate just for running eStream. If this hard minimum is greater than the soft maximum specified by the user, the hard minimum would win. I would recommend preallocating and non-zero filling the file on disk so that we know that the space is available.

Crash Resilience and Garbage Collection

In order to provide crash resilience, the index will be periodically checkpointed to disk. Note that allocating blocks does not cause problems if the index is not updated. However, we cannot reuse a page's storage until that page has been marked free on disk.

The solution to this problem is to periodically garbage-collect the cache (if it is nearly full), and writing the index to disk. The cache manager will alternate between writing two cache index files. The index file will have a marker at the end that indicates that it has been successfully written and a time stamp, and on startup the ECM will use the latest, fully written index.

Data blocks will always be written directly to the cache page files. These files must be flushed before writing the index.

Garbage collection involves the following steps:
- lock the index
- copy the free list
- choose blocks in the cache to free, and make a list containing just the newly freed blocks. Mark these blocks as invalid in the file's inodes, but don't put them on the free list (yet)

- make a copy of the index
- unlock the index
- merge the list of newly freed blocks with the copy of the free list
- flush all cache page files
- write the new, merged free list (as a bitmap) and index to disk
- lock the index
- add the newly freed blocks to the free list
- unlock the index
- free any allocated data structures

Index File Contents
The index file contains the following items:
- List of cache block files, with their sizes
- Free block bitmap, per cache block file
- Inodes for all files; may be stored hashed or may be rehashed on startup.

Testing design

Unit testing plans

Cache file manipulation routines can be tested in isolation. We will write a standalone harness that exercises the functionality of the cache file manipulation routines by performing cache level operations directly. A multithreaded unit test for the cache manipulation routines would be ideal, so we can test the correctness and performance of our locking strategy without the need to build the entire cache manager.

Each "thread" of execution described by this document can be separately tested by creating a testing harness providing that thread inputs and monitoring its outputs. Replacements for the EFSD interfaces can be very effective here.

Stress testing plans

An interesting stress test for the cache manager is if it can work correctly with very small caches, even all the way down to 1 page. (Or at least, a cache with all pages but one marked as dirty.)

The cache manager will be able to operate in "verify mode," where requests that hit in the cache will still be sent to the server, and the pages returned by the server will be compared with the cached page's contents.

The cache manager will support multiple different page checksum algorithms. We can use a fast algorithm for deployment while using a more rigorous one in development. This also has the benefit of allowing us to test the performance impact of various checksum algorithms.

The cache manager will have the ability to verify the integrity of the cache index and free page bitmap. In particular, it will have the ability to determine taht no pages are allocated to more than one file in the file system, and that each page belongs to a file or is on the free list.

Stress testing for the ECM will include crash testing.

Cache manager testing will include resizing the cache.

Coverage testing plans

Cross-component testing plans

We can build a "cache only" file system by not using the prefetching and network components. This allows us to test the EFSD in conjunction with the cache manager without involving the prefetcher or the network component.

Early implementation of the client will likely involve a null prefetcher that does no prefetching.

We can use the testing harness for the cache manager that doesn't use the EFSD to drive the cache manager in conjunction with the prefetcher and network component. This allows us to test the combination of these components without driving it with the live file system driver.

Upgrading/Supportability/Deployment design

The client user-mode software and device drivers are packaged separately. (I.e. the client executable and the drivers are separate files on the disk.) This leads to the possibility of a "partial" upgrade that results in inconsistent versions of the drivers and client user-mode software. The drivers should support an interface that returns the version number of the driver, or of the interfaces provided by the driver. This will help the client software to recognize situations where it should tell the user to reinstall the client software and not result in bad system behavior.

Most (all?) on-disk data files should have file headers containing at a minimum a magic cookie and the file format version number. This will help us with upgrades in the future.

Open Issues

We need to address what happens when a fetch is requested and no empty space can be found in the cache. The prefetcher should probably block until such time as space is

made available for this request. While operating with very small amounts of cache will obviously cause bad performance, it should not result in a deadlock.

eStream Cache Manager Straw Man Proposal

Version 0.2

[redacted]

Purpose

The purpose of this document is to serve as the basis for the design of the eStream Cache Manager. As a straw man, this document is meant to serve as the basis for discussion, and anything here is subject to change. Assuming there are no major concerns with this document, I will proceed with producing a low level design for the cache manager.

Requirements in Brief

Support > 2GB client cache, possibly across multiple drives
Provide some level of protection against piracy, via both the file system and the cache
Fast lookup for what is in the cache and where to find it
Support automatic and user-specified cache size policies

As far as cache size goes, I think that it is reasonable for eStream 1.0 for the cache to be limited to one disk partition and 2GB of space, but the design should allow for very large caches (spanning more than one file and possibly more than one drive letter.) Note that if the cache is greater than 2GB in size, it cannot be mapped into the address space of a single process under NT/2000 on x86.

Cache Organization

The cache will be contained in 2 or more files. One file will contain the cache indices, and one or more files will contain the data blocks for cached files. (More than one cache data file may be required if the cache is larger than the largest file allowed on the native file system.) This allows us to keep the cache index file memory mapped and only map the data file(s) if there is enough memory space to do so.

Data Blocks

The cache data file will contain data pages frome the file system 4k in size.

Data will be stored in the cache uncompressed to allow easy page retrieval.

Cache Index

The cache index will be a b-tree. The key for the lookup will be the file id and page number requested. Keys in the b-tree are the set { volume #, file #, starting page, # of pages }. A lookup will succeed when the volume number and file number match, and the requested page is in the range from starting page to starting page + # of pages. The data stored for that key will be the offset into the cache for the beginning of the run. As is described in the file system proposal, the file number and starting pages are each 32 bits long. I propose making the starting page a 48 bit number and the number of pages a 16 bit number. This allows us to have a very large total cache and reasonable sized runs of contiguous pages in the cache.

Free space in the cache will have to be managed. Free blocks can be placed into a specially identified "free space file" in the index. Some auxiliary data structures may be convenient to make searching for a region of free space of a particular size.

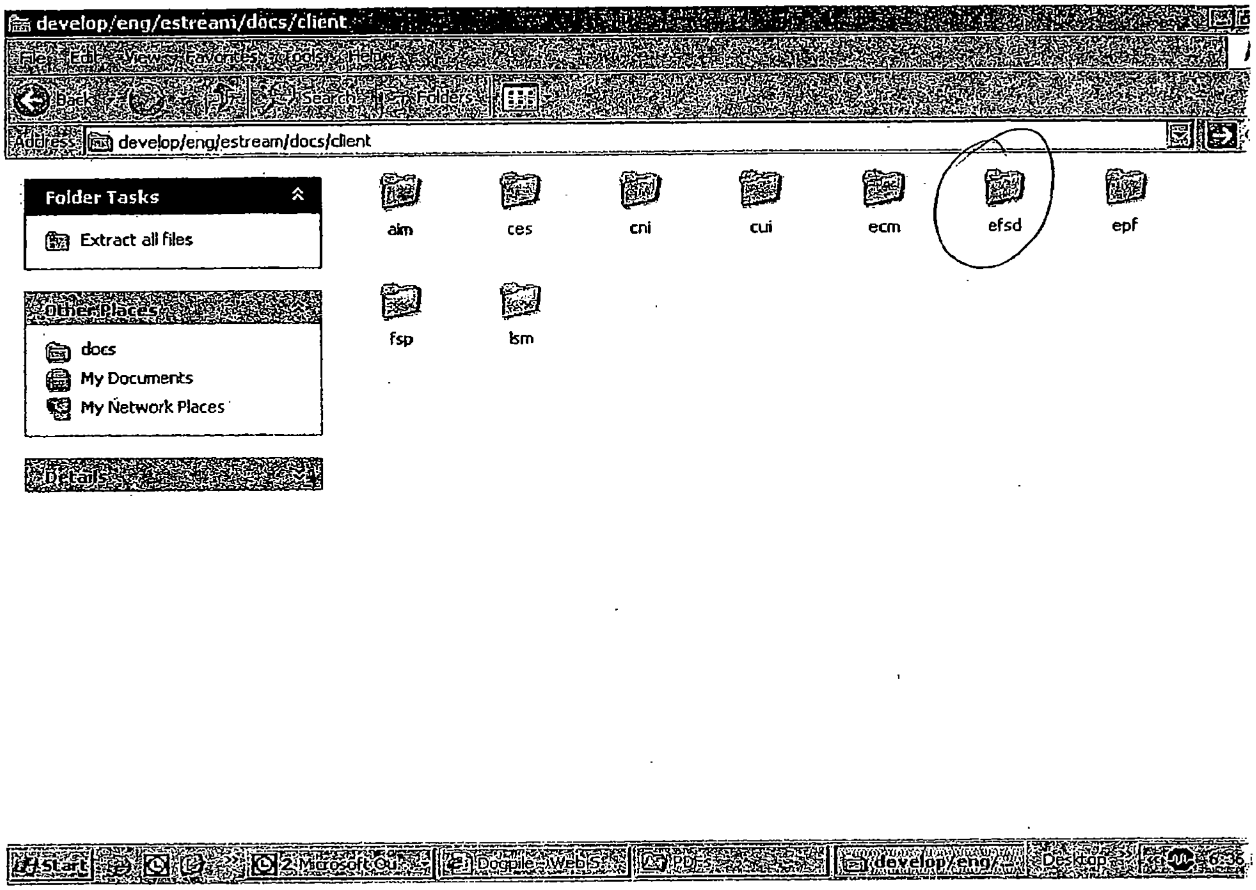
Metadata for a file would be stored in the cache. It would be indexed by page number -1 in the index.

Cache Replacement Policy
For simplicity, I propose that the cache manager evict entire files from the cache when it decides that it needs to clear room in the cache. (Of course, any fragmentary file that is in the cache can be evicted.) We should implement LRU for cache replacement, so we will evict files for apps that have not been run recently.

One Cache Per System
Administrator priviledges are required to install eStream. While various users on a system might have conflicting desires about eStream configuration, such as the size of the cache, I think that it is reasonable to have a policy where the adminstrator controls the setup of the eStream client. By limiting the cache to one per system, we eliminate any ambiguity about cache use in a multiuser environment.

Profiling and Prefetching
Profiling and prefetching have been broken out as a separate component in the client. It will be described elsewhere. It is expected that while the profiler/prefetcher will want access to the cache data structures (i.e. it wants to know what's already in the cache), the logic associated with prefetching is not logically tied to the cache manager, and should thus be separated.

Future Directions
Compression of the cache could potentially be a big win. We could provide cache compression similar to the way that NTFS provides file compression - we compress some number of blocks at a time (e.g. 16) and only store the compressed data when it saves at least one block of storage. Caching of data on disk can sometimes be a performance win, since decompressing the data can be faster than trasferring it on disk if the disk is slow enough.


develop/eng/estream/docs/client
develop/eng/estream/docs/client
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start

eStream File System Driver Low Level Design

version 1.4
Curt Wohlgemuth
[redacted]

Functionality

The eStream Windows NT/2000 File System Driver (EFSD) is a kernel-mode file system driver to which file requests will be forwarded by the NT I/O Manager. It is the point of contact for users to access files on an eStream server. It works with the NT File Cache Manager to insure that kernel file caching is available for eStreamed files.

The Windows 98 EFSD is almost certainly to be very different from the driver for WNT and Win2K, and will not be described here.

In this document, I'm assuming that the EFSD communicates closely with the eStream cache manager (ECM) to perform the various file system requests. There may in fact be several components—if for example the ECM is broken into sub-components. Also, this document assumes that the ECM is in user mode; if this ends up in kernel mode, we will need significant changes to the interfaces to it.

Data type definitions

File handle

A file handle passed between the EFSD and the ECM is defined by the ECM:

```
typedef uint32 EFSDFileHandle;
```

Names

All file and directory names will be passed as counted Unicode strings, basically as defined by the NT header files. Note, however, that in NT the Buffer field is a pointer; for our purposes in communicating with the ECM, it's a NULL-terminated variable length array:

```
typedef struct _UNICODE_STRING {
   USHORT Length;
   USHORT MaximumLength;
   USHORT Buffer[1]; // NULL-terminated, 2-byte
                     // characters
} UNICODE_STRING;
```

Time stamps

The NT standard time format is a signed 8-byte integer representing the number of 100-nanosecond intervals since [redacted] These time stamps will be tracked for files and directories:

- Creation time
- Modification time

File attributes

File attributes are contained in an unsigned 4-byte integer. This subset of attributes from Windows NT will be supported:

```
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_TEMPORARY
```

These attributes are not supported:

```
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_DEVICE
FILE_ATTRIBUTE_SPARSE_FILE
FILE_ATTRIBUTE_REPARSE_POINT
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
FILE_ATTRIBUTE_ENCRYPTED
```

File size

File size will be represented as a 4-byte unsigned integer. Since sparse files are not supported, there will only be one file size passed between the ECM and the EFSD.

Metadata

This structure is defined to pass file and directory metadata between the EFSD and the ECM:

```
typedef struct { // 24 bytes, 4-byte aligned
   int64 CreateTime;
   int64 ModifyTime;
   uint32 FileSize;
   uint32 Attributes;
```

```
} Metadata;
```

Interface definitions

The EFSD is called by several different components, including

- the NT Executive (I/O Manager, Virtual Memory Manager), for standard file system requests
- the ECM, for these same file system requests, and to invalidate cached pages for coherency
- the client start software, to start and stop the EFSD

The EFSD supports standard FSD interfaces to the NT Executive modules; not all possible interfaces are supported, because the eStream file system is relatively low-functionality (compared to NTFS, for example).

The following file system requests will be supported; the interfaces for them will not be shown here, as they can be found in the DDK documentation.

- Create IRPs, for both new and existing files
- Cleanup and Close IRPs
- Read and Write IRPs:
  - synchronous and asynchronous
  - cached and non-cached
  - paging and non-paging
- Fast I/O reads and writes (with buffers or MDLs)
- File information (get and set) IRPs
- Directory query IRPs
- Volume information (get and set) IRPs
- File system information (get and set) IRPs
- Flush buffer IRPs
- System shutdown IRPs
- Various Fast I/O queries

The EFSD will not handle Directory Notification IRPs, nor will it support hard links (which are supported natively on NTFS on W2000 only): neither of these requests are required, and no expected user functionality will be lost without them. We are presently not supporting byte-locks; this may need to be revisited if the need arises.

In addition to the interfaces to the NT Executive, the EFSD will support various interfaces from other client components; all these will be sent via IOCTL calls. The first ones listed are simple support interfaces; the interfaces between the ECM and the EFSD follow these.

An IOCTL coming in to the kernel—via a DeviceIoControl() call—has the following parameters:

- IOCTL control code
- input buffer pointer
- input buffer size
- output buffer pointer
- output buffer size
- pointer to a 4-byte variable to receive the number of bytes written to the output buffer
- pointer to an OVERLAPPED structure for asynchronous operation (always should be NULL for EFSD)

All of the following interfaces are described in terms of the IOCTL buffers sent and received for each control code.

The following interfaces are called from the controlling client component (StartClient).

Starting and stopping the file system

The eStream FSD will be loaded into the kernel when a system is rebooted; i.e., it is always resident. If applications access files on this FSD via a drive letter, then the file system is implicitly turned off while a symbolic link for that drive letter is not present. Even when a drive letter symlink exists, the EFSD will not accept requests until the START IOCTL is sent.

These IOCTL control codes will be defined for starting and stopping the eStream FSD:

```
IOCTL_EFS_START_FS
IOCTL_EFS_STOP_FS
```

Starting the FSD

The input buffer for the START IOCTL should have the following:

- version id: 4-byte identifier for the client component
- debug flags: 4-byte value indicating the debug level to use

The output buffer for this IOCTL will be filled with the following:

- version id: 4-byte identifier for the EFSD version present
- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BAD_VERSION
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_DUPLICATE_REQUEST
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

A DUPLICATE_REQUEST error is returned if the FSD is already started.

Stopping the FSD

The input buffer for the STOP IOCTL should have the following:

- force: 4-byte value
  - 0: shutdown only if no outstanding files are open
  - 1: shutdown regardless of state of open files

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_DUPLICATE_REQUEST
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

A DUPLICATE_REQUEST error is returned if the FSD is already stopped.

Cache management interfaces

The following two interfaces are defined for use by the ECM to potentially invalidate data in the NT File Cache.

These IOCTL control codes will be defined for cache management for the eStream FSD:

```
IOCTL_EFS_INVALIDATE_FILE
IOCTL_EFS_INVALIDATE_DIR_CONTENTS
```

Invalidating a file

The input buffer for the INVALIDATE_FILE IOCTL should have the following:

- handle: 4-byte EFSDFileHandle for the open file that must be invalidated

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_FILE_NOT_OPEN
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

If in fact the file is open, but not present in the NT File Cache, this IOCTL will simply succeed; no error is returned.

Invalidating directory contents

The input buffer for the INVALIDATE_DIR_CONTENTS IOCTL should have the following:

- handle: 4-byte EFSDFileHandle for the open directory whose contents must be invalidated

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_FILE_NOT_OPEN
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

General file system requests

All file system requests that cannot be completely handled by the EFSD will be passed on to the ECM. Since the ECM is likely to be a user-mode service, the EFSD cannot call it directly; thus these "calls" are made by having the ECM send IOCTLs to the EFSD to get and fulfill requests. Each file system request requires multiple IOCTLs sent from the ECM to the EFSD:

1. The ECM sends an IOCTL to the EFSD to get the next request
2. The ECM sends a second and/or third IOCTL to finish the request

The following IOCTL control codes will be defined by the EFSD for use by the ECM:

```
IOCTL_EFS_GET_REQUEST
IOCTL_EFS_RETRY_REQUEST
IOCTL_EFS_GET_CREATE_NAME
IOCTL_EFS_FINISH_CREATE
IOCTL_EFS_FINISH_CLOSE
IOCTL_EFS_FINISH_READ
IOCTL_EFS_GET_WRITE_DATA
IOCTL_EFS_FINISH_WRITE
IOCTL_EFS_GET_RENAME_TARGET
IOCTL_EFS_FINISH_RENAME
IOCTL_EFS_FINISH_DELETE
IOCTL_EFS_FINISH_METADATA_READ
IOCTL_EFS_FINISH_METADATA_WRITE
```

For the DeviceIoControl() call sending IOCTL_EFS_GET_REQUEST, these parameters are invariant:

- the IO control code will be IOCTL_EFS_GET_REQUEST
- input buffer pointer will be NULL
- input buffer size will be 0
- output buffer must be non-NULL
- output buffer size must be **at least 40 bytes**—this is the largest buffer needed for any request (subject, of course, to slight modifications)
- pointer to bytes returned will be non-NULL
- overlapped pointer will be NULL

The IOCTL_EFS_RETRY_REQUEST is sent by the ECM (or some other user-space client component) to tell the EFSD that, yes, it needs to delete all intermediate information about a request already sent back with a GET_REQUEST call, and put the request back on the list for the ECM to retrieve. This eases implementation issues for the ECM. The input buffer for a RETRY_REQUEST is:

- request id of the previously retrieved request

There is no output buffer for a retry request IOCTL.

What follows is a list of file system requests from the NT I/O Manager, and the IOCTL calls needed from the ECM to service those requests. For all cases, if the EFSD writes to the output buffer for an IOCTL, the "bytes returned" field is written with the number of bytes written.

Create

This is used for both create and open, for files and directories.

GET_REQUEST

The output buffer for the IOCTL_EFS_GET_REQUEST will be filled with the following:

- type: a 4 byte field that indicates a Create request
- request id: a 4 byte field that will be subsequently sent in the calls to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- flags: 4 bytes, one or more of the following ORed together
  - CREATE_ONLY: fail if file exists already
  - OPEN_ONLY: fail if file does not exist already
  - TRUNCATE: overwrite existing file
  - DIRECTORY: create a directory
  - FILE: create a plain file
  - DELETE_ON_CLOSE: delete file on last close
  - IGNORE_CASE: obvious
- permissions: 4 bytes, one or more of the following ORed together
  - READ
  - WRITE
  - EXECUTE
- length of filename: 4 bytes, specifying the byte size needed for the Unicode string sent in the next call

Total size of output buffer: 24 bytes

GET_CREATE_NAME

The input buffer for this IOCTL should have the following data:

- request id: the id sent in the previous call

The output buffer for this IOCTL will be filled with the following information:

- request id for this transaction
- fully qualified name as a counted Unicode string (including drive letter, if any): the length needed was sent back in the GET_REQUEST call

FINISH_CREATE

The input buffer for this call should have the input buffer filled as follows:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- handle: the 4-byte handle for this opened file, that can be used for subsequent file system requests. A unique value will indicate a bad handle, and a failed Create
- a Metadata buffer: the metadata for the created/opened file/directory.

The output buffer for this IOCTL should be NULL.

Note that a TRUNCATE Create request should cause the metadata sent back to reflect the possibly new (zero) length.

Close

This closes a handle of a previously opened file or directory. The EFSD will optionally send the updated metadata for this file in the GET_REQUEST output buffer. If the file has been modified in any way, the metadata fields will be non-zero; else they will all be zero.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Close request
- request id: 4 bytes, for use in subsequent calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for the previously opened file
- metadata for this file/directory: 24 bytes
    - creation time stamp
    - modification time stamp
    - file/directory size in bytes
    - attributes (as described above)

Total size of output buffer: 40 bytes

FINISH_CLOSE

The input buffer for this call should contain the following:

- request id for this transaction
- status: the NTSTATUS for this request

Read

This is used for reading file data.

GET_REQUEST

The output buffer for the IOCTL_EFS_GET_REQUEST will be filled with the following:

- type: a 4 byte field that indicates a Read request
- request id: a 4 byte field that will be subsequently sent in the IOCTL to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for this previously opened file
- offset: 4 bytes, the file offset, in bytes, to read from
- length: 4 bytes, the length of the read, in bytes

NOTE: The buffer requested in the (offset, length) pair will *not* cross a 4K page boundary.

Total size of output buffer: 24 bytes.

FINISH_READ

The input buffer for this call should have the input buffer filled as follows:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- the number of bytes successfully read; 0 on error
- the data from the read; not present on error

The output buffer for this IOCTL should be NULL.

Write

This is used for writing file data.

GET_REQUEST

The output buffer for this will be filled with the following:

- type: a 4 byte field that indicates a Write request
- request id: a 4 byte field that will be subsequently sent in matching calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for this previously opened file
- offset: 4 bytes, the file offset, in bytes, to write to
- length: 4 bytes, the length of the write, in bytes
- file length: 4 bytes, the length the file will be if this write succeeds

Total size of output buffer: 28 bytes.

NOTE: The buffer requested in the (offset, length) pair will *not* cross a 4K page boundary.

GET_WRITE_DATA

This IOCTL will have an input buffer with:

- request id for this transaction
- status: if not STATUS_SUCCESS, this ends the request; the output buffer is untouched, and no FINISH_WRITE call is expected.

And the output buffer will be filled with:

- request id
- data buffer for the write—the byte length sent in the previous GET_REQUEST

FINISH_WRITE

For this finishing request IOCTL, the input buffer has these contents:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- bytes actually written; should be equal to requested bytes unless failure occurs.

Rename

This is used for renaming a file or directory.

GET_REQUEST

The output buffer for this IOCTL will be filled with the following:

- type: a 4 byte field that indicates a Rename request
- request id: a 4 byte field that will be subsequently sent in the FINISH_REQUEST call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory
- length of target name: 4 bytes; the byte length needed for a counted Unicode string for the target name

Total size of output buffer: 20 bytes.

GET_RENAME_TARGET

The input buffer for this call will have the following:

- request id for this transaction
- status: if not STATUS_SUCCESS, then this terminates the request: the output buffer is not touched, and no FINISH_RENAME call should be sent

The output buffer will be filled with the following:

- request id
- target name: a counted Unicode string, using the same number of bytes as sent in the GET_REQUEST output buffer

FINISH_RENAME

The input buffer for this call should have the following:

- request id
- status: NTSTATUS for the transaction

Delete

This is used for deleting a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Delete request
- request id: a 4 byte field that will be subsequently sent in the call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory

Total size of output buffer: 16 bytes.

FINISH_DELETE

The output buffer for this call should be NULL.

The input buffer should have the following contents:

- request id: matching id from the GET_REQUEST call
- status: NTSTATUS of this request

Metadata read

This is used for requesting metadata about a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Metadata request
- request id: a 4 byte field that will be subsequently sent in the call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory

Total size of output buffer: 16 bytes.

FINISH_METADATA_READ

The output buffer for this IOCTL will be NULL.

The input buffer should have the following contents:

- request id: id from the corresponding GET_REQUEST
- status: NTSTATUS for this operation
- the following data about the file or directory:
  - creation time stamp
  - modification time stamp
  - file/directory size in bytes
  - attributes (as described above)

Metadata write

This is used for setting metadata for a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Metadata Write request
- request id: a 4 byte field that will be subsequently used for all calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory
- metadata for this file/directory: 24 bytes
  - creation time stamp
  - modification time stamp

- o file/directory size in bytes
- o attributes (as described above)

Total size of output buffer: 40 bytes.

FINISH_METADATA_WRITE

The input buffer should have the following contents:

- request id: from the previous call
- status: NTSTATUS for this request

Component design

This section is organized in the following manner:

1. General layout of the eStream file system driver
2. General observations about the low level design
3. Organization of data structures
4. Description of the algorithms for communication with the ECM
5. Description of each dispatch routine

Layout

The EFSD will be generally organized in the following manner:

- All major IRPs will have their own dispatch routine.
- All actual I/O requests to the ECM will be generalized from the dispatch routines to a set of routines that handle the communication with the ECM, to isolate this aspect.
- All utility functions will be in their own file or files.

General points

The design of the EFSD will look a lot like the sample FSD from Rajeev Nagar's NT FS Internals book, which looks a whole lot like the Fastfat FSD source from the NT IFS kit.

Here is a list of general points that can be made about the EFSD:

- Any IRP that can be handled asynchronously will be posted to a work queue; this means that the dispatch routine for such an IRP must be able to handle being called in a context other than the original requestor.
- There are no volumes, and no Volume Parameter Block or Volume Control Block. There isn't a VPB for a network redirector; I've verified this with the

LanManager redirector. Hence we don't have to support any operations on a volume in EFS.

- We will not allow the creation of paging files in EFS. There is a bit available for a Create IRP that specifies this, and we can complete the IRP with an unimplemented error return code.
- All file synchronization will be on a File Control Block (FCB) basis, using the standard Resource and PagingIoResource ERESOURCE objects used by the rest of the Windows Executive.
  - User requests will be synchronized by acquiring the main Resource—shared for reads, shared for most writes, and exclusive for file size changes, deletion, etc.
  - Paging I/O requests will be synchronized by acquiring the PagingIoResource—again, shared for reads, shared for most writes. Exclusive access will be needed to set file sizes.
- Most disk file systems have a resource associated with a VCB, which is acquired exclusively for creation/deletion etc. We will have a global EFS resource for this, since there are no VCBs.
- Asynchronous requests will be handled by posting the IRP to the CriticalWorkQueue, and marking the IRP as pending.
  - A common worker routine will be used for all async posts, which will dispatch the IRP to the appropriate real IRP routine when it's invoked.
  - An async request will be defined as one that IoIsOperationSynchronous() returns false, and the EFSD is the top-level component (see below)
- The EFSD will track the top-level IRP for the thread whose context it is running in. In particular,
  - No async processing request will be honored unless the EFSD is the top-level component
  - No cache manager requests will be made unless the EFSD is the top-level component
  - EndOfFile size—that is, the true size of the file—will not be extended or changed by paging I/O
- EFS will not support holes in files, and hence the ValidDataLength FCB field will be set to disable this. This means the AllocationSize for an eStream file/directory will always be equal to the EOF size.
- Most fast I/O routines will be supported in EFS. We will use the FSRTL supplied routines for fast reads and writes.
- All cache manager resource acquire/release callbacks will be supported. All will point to common routines that simply acquire or release the main or paging I/O resource for the FCB. The Context pointer passed into all of them will be the FCB for the stream.
- Synchronous read/write requests will update the CurrentByteOffset in the File object.
- Each Create will result in a unique Context Control Block (CCB) data structure; this will be small, and only hold those few fields needed:
  - For the Directory Control IRP, a CCB needs to hold the current entry index and the pattern originally used—for subsequent queries

  - A field for various flags
- A single FCB will represent all current open instances of a file. When a Create request comes in, the EFSD will search the current open FCBs to try to find one matching this file/directory name.
  - For now, this will be a hash table on the file name. We can improve this as needed.
  - The EFS global resource must be acquired exclusively:
    - before the global FCB data structure is searched. **Why? If it's just a read, can't we acquire it non-exclusively?**
    - before a new FCB is added to the list
    - before an FCB is deleted from the list
- EFS will not support open by file ID; hence the FileInternalInformation class for a File Information IRP will not be supported.
- Actual I/O will be directed to standard routines in a separate file, so they can be isolated and updated easily as our method of transferring data changes.
- Here's how to do file/directory renames:
  - The I/O manager will send to the EFSD this sequence:
    - Create for source
    - Create for target, with the SL_OPEN_TARGET_DIRECTORY flag set
    - Set Information with a Rename request for the source, sending the target directory FileObject handle and the target name in the FileInformationClass record.
  - EFSD needs to do this:
    - When it receives the Create for the target and the target *directory* exists, return STATUS_SUCCESS, and change the name in the FileObject to the basename of the target (the full pathname of the target is sent in), and set the Status.Information to FILE EXISTS or FILE DOES NOT EXIST, as appropriate. If the target directory doesn't even exist, return PATH NOT FOUND.
    - When it receives the Set Info request, if all the flags check out (e.g., if the file exists, ReplaceExisting must be TRUE), send a Rename request to the ECM.
- Reads and writes to only regular files will be supported, not to directories.
- Any code that touches user buffers or can call routines that may throw exceptions must be guarded by a try/except block.
- Some tips on memory allocation (from /perforce-doc-dir/osrdocs/defensive-driv.html)
  - We will use our own memory allocation/deallocation routines, instead of ExAllocatePool() et al. directly
  - These routines can do various checks for trashing memory:
    - fill allocated memory with a pre-defined bit pattern, instead of zeroes; fill deallocated memory with a different pattern.
    - allocate a header/trailer with standard information, like where allocated, from what pool, etc.

        - change the bit pattern in the header/trailer on deallocation, and look for freeing memory twice
    - We probably want to allocate using lookaside lists, since we'll be allocating and deallocating smallish chunks of memory for our data structures.

Data structures

The following are major data structures used internally by the EFSD. Data structures used to communicate with the ECM are described in the following section.

NodeIdentifier

A NodeIdentifier is a simple structure that starts all other structures used in the EFSD. This makes a good debugging check to insure that we receive and are operating on the right type of data. It consists of two fields: an identifier field, and the total structure size.

FCB

The FCB is a critical data structure for the file system driver. There is one FCB structure allocated for each unique file or directory that is currently open—regardless of how many open handles there are for this entry. Multiple CCB structures can point to a single FCB.

Logically at least, part of an FCB is exposed to the Cache Manager and VMM to support caching and paging I/O. We will follow the example of Rajeev Nagar's book, and embed the FSRTL_COMMON_FCB_HEADER and other required structures directly in an FCB.

Here are the basic contents of an EFSD FCB:

- The required FCB contents above
- An open handle count: incremented on Create, decremented on Cleanup
- A reference count: incremented on Create, decremented on Close. The semantics of NT file requests require these two be used together to determine when an FCB can be deleted
- A pointer to the next FCB on its hash list
- A pointer to the first CCB opened for this FCB
- The fully qualified name of the file/directory
- The Metadata associated with this file/directory
- A SHARE_ACCESS structure used to check sharing violations
- Various flag bits

CCB

A CCB represents a currently opened handle to a file or directory. If two processes have the same file opened, there will be a unique CCB allocated for each process.

Here are the basic contents of an EFSD CCB:

- A pointer to its corresponding FCB
- A pointer to the next CCB opened for its FCB
- A pointer to the FileObject opened for this file/directory
- The current file index for a directory query
- The directory search pattern used for directory queries for this opened handle
- Various flag bits needed

IrpContext

An IrpContext structure encapsulates the interesting data from an IRP, and the current stack location, for easy access. This structure is allocated on entry to dispatch routines, and used during processing, before being deallocated on exit.

Communicating with the ECM

I tend to divide a file system driver into two logical parts:

1. A front-end that understands the NT FSD interfaces and semantics
2. A back-end that actually perform the requested actions

Our back-end is the (admittedly ugly) interface for communicating with the ECM, which currently sits in user-space. It's very important to design the ECM such that its front-end and back-end are nicely separated: since the ECM may move to kernel space, or we might find better interfaces for them to communicate with each other, we need to design with this in mind.

Given the current ECM interfaces defined above, here is a basic design to handle them:

- An EfsdRequest object will be created for each request that must be handled by the ECM.
- Each dispatch routine that results in a request to the ECM will allocate an EfsdRequest and send it to a common routine for further processing.
- New requests will be placed in a NewRequest queue.
- Requests that have been "sent" to the ECM, but not yet finished, will be removed from the NewRequest queue and placed in a PendingRequest list.
- Finishing a request entails removing it from the PendingRequest list, returning the contents to the dispatch routine, and destroying the request object.

Data structures

EfsdRequest

This contains:

- request id: a number that uniquely identifies this request
- type: the type of request (e.g., Create, Write)
- FCB pointer for the file/directory for this request
- IrpContext pointer for this request
- a kernel event object used for signaling that the request is satisfied

NewRequestSemaphore

The EFSD device object's device extension will contain a semaphore dispatcher object, initialized to non-signaled, and with an initial limit of MAX_LONG. When a request is added to the NewRequest queue, this semaphore is released (and the count is incremented by 1); the GET_REQUEST IOCTL will wait on this semaphore object (which decrements the count by 1).

NewRequestQueue

This is actually a kernel-managed interlocked list that is allocated in the device extension area, guarded by a spin lock that's also located in the device extension. Requests will be added to the tail, retrieved from the head.

PendingRequestList

This list must be searched when an ECM non-GET_REQUEST IOCTL is received, so we can't use an interlocked list. We'll use a global single-linked list structure, with elements allocated from a dedicated lookaside list using non-paged memory. Before a thread can access the list elements, it must acquire a mutex.

Algorithm

- All dispatch routines, and asynchronous read/write routines, will call LowLevelPostRequest() to have their requests satisfied. LowLevelPostRequest() is itself synchronous; that is, the code calling it is something like this:

```
status = LowLevelPostRequest(fcb, irp_context);
      // we're done;
      // do all deallocation and cleanup needed
IoCompleteRequest(irp, ...);
```

- LowLevelPostRequest() will do the following:

```
if this is a read or write IrpContext
    see how many requests are needed to satisfy the IRP: can't span page boundary
    allocate the N requests
    allocate an array of N event pointers to hold the request events
    assign the pointers to the array
    if > THREAD_WAIT_OBJECTS
        allocate an array of N PKWAIT_BLOCKS
```

```
else if this is a directory query IrpContext
    look at the FileSize of the directory FCB
    allocate enough read requests to read all directory data from the ECM
    as above, allocate an array of event pointers, wait objects (if necessary)
else
    allocate a new EfsdRequest for the incoming FCB and IrpContext
place all requests on the NewRequestQueue, using ExInterlockedInsertTailList()
call KeReleaseSemaphore() on the NewRequestSemaphore
if multiple requests generated
    call KeWaitForMultipleObjects() on the request object's event
else
    do a KeWaitForSingleObject() on the request object's event
when the event(s) is/are signaled
    fill in the values into the IRP
    deallocate the EfsdRequest(s)
    return status
```

- When a GET_REQUEST IOCTL comes in from the ECM, the EFSD will do the following:

```
do a KeWaitForSingleObject() on the NewRequestSemaphore
remove the first request from the NewRequestQueue,
                    using ExInterlockedRemoveHeadList()
lock the PendingRequestList mutex
place this request on this list
release the mutex
fill in the IOCTL output buffer identifying the request
complete the IOCTL IRP
```

- When a RETRY_REQUEST IOCTL is received, the following takes place:

```
lock the PendingRequestList mutex
search the list by request id; error if not found
remove the request from the list
release the mutex
enqueue the request on the NewRequestQueue—on the list head
release the NewRequestSemaphore
complete the IOCTL IRP
```

- When any "finishing" IOCTL is received—i.e., the second of two or the third of three—the following is done:

```
acquire the PendingRequestList mutex
search the list by request id; error if not found
remove the request from the list
release the mutex
do all buffer copying, set all flags,
            and otherwise insure the input IRP has the correct state
signal the EfsdRequest event
complete the IOCTL IRP
```

- When any other request IOCTL is received—e.g., the second of three—this is done:

```
acquire the PendingRequestList mutex
search the list by request id; error if not found
release the mutex
fill in the output buffer of the IOCTL as appropriate
complete the IOCTL IRP
```

Dispatch routines

DriverEntry

This does a whole slew of initialization, including the dispatch table, fast I/O table, the cache callbacks, the FCB hash table and its synchronization object, creates the FS device object, and sets up the interface with the ECM.

Create

There is one Create routine; there will be no async processing of Create requests. Ultimately, its job is to send a create request to the ECM, and return SUCCESS or not to its caller. Here is a general algorithm for this routine.

```
create an IrpContext
if a page file is requested
    return error
generate the absolute pathname of the requested file:
    if OPEN_TARGET_DIRECTORY specified
        if OPEN_ONLY not specified
            return error
        generate the pathname of the parent directory of the requested file
    if a related file object is specified
        if the related file is not a directory
            return error
        if the related filename doesn't start with '/'
            return error
        if the input filename starts with '/'
            return error
        concatenate the input filename with the related file directory
    else use the input filename
        if the input filename does not start with '/'
            return error
acquire the global EFS resource exclusively
search the FCB hash table for this file by name
if not found
    call LowLevelPostRequest() to send the request to the ECM
    if error is returned from ECM
        return the correct error: FILE NOT FOUND or PATH NOT FOUND
    create a new FCB and add to the hash table
    call IoSetShareAccess() for this FCB
else
    check input attributes/flags against those in FCB
        if opening for write or delete on close
            call MmFlushImageSection()
            if this fails
```

```
            return error
      if failing mismatch—e.g., if IoCheckShareAccess() fails
            return error
create a new CCB for this file
set all appropriate flags on CCB and/or FCB
      COMMON_FCB_HEADER flag is set to FastIoIsPossible
set all fields on input FileObject:
      write through flag
      FsContext points to common FCB header
      FsContext2 points to CCB
if OPEN_TARGET_DIRECTORY is specified
      search for the input target object:
            look for this in the FCB hash table
            if not found
                  send a Create request for this file to the ECM
      if this target filename exists
            set Status.Information to FILE_EXISTS
      else
            set Status.Information to FILE_DOES_NOT_EXIST
      if a Create was sent to the ECM for the input FileObject
            send a Close request for the input target to the ECM
      change the name in the FileObject to the basename of this file
      the CCB and FCB remain opened for the target directory
all necessary data structures should be deallocated, for success and error
release the global EFS resource
```

Cleanup

No async posting of Cleanup requests will be done. Algorithm:

```
acquire the global EFS resource, and the FCB Resource, for this file, exclusively
if this file is marked for deletion
      if this is the last open handle for the file
            acquire the PagingIoResource exclusively
            set the file size in the FCB to 0
            release the PagingIoResource
            purge the cache, if necessary, with MmFlushImageSection()
            call LowLevelPostRequest() to send a Delete request to the ECM
decrement the count of open handles in the FCB
if caching is on
      flush the cache by calling CcUnitializeCacheMaps()
any time stamps must be updated if accesses were done using fast I/O
set the FO_CLEANUP_COMPLETE flag in the FileObject
call IoRemoveShareAccess()
release the global EFS and FCB Resources
```

Close

There will be no async posting of Close requests. Note that we only send a Close request to the ECM if this is the last close for an open file—i.e., we're matching Close requests with Create requests.

Algorithm:

```
acquire the global EFS resource exclusively and the FCB Resource
deallocate the CCB
decrement the reference count for the FCB
if the FCB ref count is now 0
    remove the FCB from the hash table
    deallocate the FCB
    call LowLevelPostRequest() to send a Close request to the ECM,
                                            updating metadata if necessary
release the global EFS resource and the FCB Resource
```

Read

Reads will definitely be open to async posting. A general algorithm:

```
if the read length is 0
    return success
if the target file object is a directory
    return error

if this IRP can be handled async
    post for async processing

if this read is non-buffered, and it's not for paging I/O, and
                        there is a mapped data section object for this file
    acquire the FCB main Resource exclusive, and the PagingIoResource shared
    call CcCacheFlush() on the range of this read (current byte offset, length)
    release the FCB resources
if this is for paging I/O
    acquire the PagingIoResource shared
else
    acquire the main Resource shared
if this read starts beyond EOF
    return EOF
if the read length goes beyond EOF
    truncate the length
if this is a buffered read
    if the PrivateCacheMap is NULL
        call CcInitializeCacheMap()
    if this is an MDL read
        call CcReadMdl()
    else
        call CcCopyRead(), using either an allocated MDL or the UserBuffer

else (it's non-buffered)
    call LowLevelPostRequest() to send a Read request to the ECM

if this is a synchronous, non-paging read
    update the current byte offset in the FileObject
set the number of bytes read in the Status.Information field
release any acquired resource and deallocate appropriate data structures
```

Write

Writes will definitely be open to async posting. A general algorithm:

```
if the write length is 0
    return success
if the input file is a directory
    return error
if the file not opened with write permissions
    return error

if this IRP can be processed asynchronously
    post for async processing

if this is a buffered write
    call CcCanIWrite()
    if false
        we have a hard error; fail
if this is paging I/O
    acquire the PagingIoResource shared
else
    acquire the main Resource shared
if the length is
        (Low == FILE_WRITE_TO_END_OF_FILE) && (High == 0xffffffff)
    we're to write at EOF
if this is a non-paging, non-buffered write, and
                                there is a mapped data section object for this file
    acquire the global EFS resource exclusive
    acquire the PagingIoResource shared, starving exclusive waiters
    call CcCacheFlush() on the range for this write
    release the PagingIoResource
    return error if the cache flush failed
    acquire and release the PagingIoResource exclusive (to serialize)
    call CcPurgeCacheSection() on the range for this write
    release the global EFS resource

if this is a paging I/O write
    if the starting offset is beyond the current EOF
        return success
    if the ending offset is beyond the current EOF
        truncate the write length to EOF

if this is a buffered write
    if the private cache map is NULL
        call CcInitializeCacheMap() for this file
    if the write will extend the file size
        release the resource acquired
        re-acquire the resource exclusive
        call CcSetFileSizes() inform the cache manager
    if this is an MDL write
        call CcPrepareMdlWrite()
    else
        call CcCopyWrite(), using either an allocated MDL or the UserBuffer

else
    call LowLevelPostRequest() to send the write request to the ECM

set the number of bytes written in the Status.Information field
update the file size fields in the FCB if this write extends the length
if this is a non-paging write
```

```
        set the CCB modification flag
        if this write is synchronous
            update the CurrentByteOffset field in the FileObject
    release any acquired resource and deallocate appropriate data structures
```

Fast I/O Read

Initially at least, we'll just set the fast I/O read routine to FsRtlCopyRead().

Fast I/O Write

Initially at least, we'll just set the fast I/O write routine to FsRtlCopyWrite().

Fast I/O Query Basic Info

This will just fill in the basic info buffer with the data in the FCB.

Fast I/O Query Standard Info

This will just fill in the standard info buffer with the data in the FCB.

Fast I/O Query Open

This will just fill in the network open info buffer with the data in the FCB, if the file exists. Some empirical observations I've made using NTFS:

- Regardless of whether the file exists or not, this will return TRUE (all fast I/O routines are boolean)
- If the file does not exist, it will set the EOF size in the buffer to 0. The AllocationSize must be non-zero. All other fields seem to be don't cares.
- If the file exists but is zero length, then both EOF and AllocationSize will be 0.
- The IRP sent to this routine is for an IRP_MJ_CREATE; we can use more than just the name to identify the file, but also the security characteristics or whatever else is sent in the IRP.

File Query Info

Standard queries will be supported; these however will not:

- FileInternalInformation—no OPEN_BY_FILE_ID
- FileEaInformation—no EA data
- FileCompressionInformation—no on-disk compression
- FileStreamInformation—no multiple streams

File Set Info

These actions will be supported:

- EndOfFile size changes
- AllocationSize changes
- Time stamp changes
- File position changes
- File disposition changes—delete pending
- File rename requests

Here is a general algorithm:

```
if AllocationSize or EOF size change
    if the new size is smaller than the current size
        call MmCanFileBeTruncated(), to ask the VMM if this is okay
        if yes
            call CcSetFileSizes()
        else
            return STATUS_USER_MAPPED_FILE error
    else
        send a Metadata Set request to the ECM with new, extended size
        if status returned is error
            return error (disk space full)
    update AllocationSize and FileSize fields of required FCB header

else if time stamp change
    send Metadata Set request to ECM
    if error returned
        return with error

else if file position change
    update FileObject CurrentByteOffset field

else if disposition (delete) change
    if the Delete flag in the IRP not set
        clear delete on close FCB flag
    if file already marked for delete on close
        return success
    if file not open with write permission
        return error
    if MmFlushImageSection() fails
        return error
    if this is a directory, and the directory is not empty
        return error
    set delete on close flag in FCB

else if rename change
    if the source name is for a directory
        if the directory has any open files/directories under it
            return error

    if Parameters.SetFile.FileObject is NULL (simple rename)
        target dirname is the same as dirname of IRP FileObject
        target basename is Buffer.FileName

    else
        if Buffer.RootDirectory is NULL (fully-qualified rename)
```

```
            fully qualified target name is Buffer.FileName

        else  (relative rename)
            call ObReferenceObjectByHandle() to get the file object of the relative directory
            from file object, get (fully qualified) dirname of target
            append Buffer.Filename to root directory dirname to get fully qualified target

    if the target name isn't on EFS
        return error
    if target exists
        if Parameters.SetFile.ReplaceIfExists is FALSE
            return error
        if target is a directory
            return error
        if target is read-only
            return error
        if target has any open handles to it
            return error

    call LowLevelPostRequest() to send a Rename request to the ECM
```

Directory Query

Directory queries turn into directory read requests to the ECM. The EFSD will take the contents of the read buffers and fill the IRP_MN_QUERY_DIRECTORY buffers sent to it from the NT Executive by parsing the directory contents.

Note: this design does not use the NT Cache Manager for metadata or for directory entries, nor does the EFSD store the directory contents anywhere in its data structures. It will always go back to the ECM for directory queries. Given that most directory queries occur in this order:

Create
Directory query
Close

unless we can cache the contents in a location more persistent than an FCB, we will need to resubmit the request to the ECM for each new directory query. If this poses a performance problem, we need to handle it then.

Directory queries are subject to async processing.

This is an ugly NT interface. Here are some general points regarding directory queries, and how they will be implemented on EFS:

- These requests come in from the I/O Manager in a context-sensitive sequence. I.e., a request will come for the initial N directory entries; the next request will be for the next M entries; etc. Kind of like strtok().
- Thus, state must be maintained from request to request. This state will be kept in the CCB for a file stream, and consists of:

  - Pattern sent in on first request
  - Index of n'th entry to start retrieving with
- My experience is that the INDEX_SPECIFIED flag is **never** set in a directory control query, even on queries subsequent to the initial one.
- For EFSD, the FileIndex will represent the offset of the fixed-length portion of a directory entry as returned by the ECM.
- Initially, at least, **all** directory queries will cause the EFSD to read all the entries for that directory from the ECM. We can modify this later if needed.

Here is a general algorithm, based on the sources for the Fastfat driver:

```
if the FileObject is not for a directory
    return error

post this for async handling, to read all directory pages from the ECM

if the CCB pattern field is empty and the CCB flag "match all" isn't set
    this is the initial query
    acquire the FCB Resource exclusive
else
    acquire the FCB Resource shared

get a pointer to the input buffer, using either an MDL or the UserBuffer
if this is the initial query
    parse the FileName pattern
    save the pattern in the CCB
    if the pattern is "*"
        set the "match all" flag in the CCB

if SL_RESTART_SCAN is specified
    use an index of 0 for the query
else if SL_INDEX_SPECIFIED is specified
    use the input index for the query
else
    if this is the initial query
        use an index of 0
    else
        use the index saved in the CCB

start with the directory entry corresponding to the starting index
if this index is beyond the number of entries in the directory, and this is not the initial query
    return STATUS_NO_MORE_FILES
while there are more directory entries
    if the directory entry name matches the pattern in the CCB,
                                        using FsRtlIsNameInExpression()
        if there isn't room in the buffer for this entry
            break
        write the entry in the input buffer
        if there is a next directory entry beyond the current one
            the FileIndex field is set to the offset of this next directory entry
        else
            the FileIndex field is set to 0
        if there is a previously written entry
            fix up the NextEntryOffset in the previous entry to the byte offset of this entry
```

```
        if SL_RETURN_SINGLE_ENTRY specified
            break
        8-byte align the current pointer in the user buffer
    advance to next directory entry

if we wrote nothing
    if we stopped because of no room
        return STATUS_BUFFER_OVERFLOW
    else
        return STATUS_NO_SUCH_FILE

update the index field in the CCB
Status.Information is set to the number of bytes written
return STATUS_SUCCESS
```

File System Query Info

Empirically, I've noticed that the LanMan redirector returns failure for most of these requests. So, except for any user-defined FSCTL requests we want to define, I'm going to fail all of these until it turns out we need to do otherwise.

File System Set Info

Ditto for this IRP type too.

Volume Query Info

We at least need to minimally implement these requests:

- FileFsAttributeInformation
- FileFsVolumeInformation
- FileFsDeviceInformation

This will be handled solely by the EFSD; the request will not go out to the ECM. File system size requests will not be handled.

Volume Set Info

We will fail all requests of this type.

Flush Buffers

A buffer flush request for a file stream will mean the following:

- If the file stream isn't buffered, return immediately
- The FCB main Resource is acquired exclusive
- The Cache Manager is told to flush the buffer for the byte range of the file
- The resource is released

A buffer flush for a directory is a successful NOP.

The buffer flush request will insure that contents in the NT File Cache are written to the ECM (as a normal write request); the buffer flush request itself will not be propagated to the ECM.

Testing design

Unit testing plans

The EFSD will be tested apart from integrating with the ECM or the rest of the eStream client. Some points:

- There will be a (relatively) simple stand-in user process for the ECM, to get requests from the EFSD and handle them locally.
- As much EFSD functionality as possible will be done using user-mode test cases (e.g., open files, read/write files, delete files, etc.).
- Some functionality may need to be unit tested using another kernel-mode driver to send explicit IRPs to the EFSD.
- Filemon will be used to monitor the requests being handled by the EFSD.

Stress testing plans

I've heard of a file system filter driver test package available from Microsoft. This is probably the best way we have of stress testing the EFSD.

Coverage testing plans

We'll try to measure coverage on the EFSD. If there is a general kernel-mode method for measuring coverage that's used company-wide, we'll exploit it. Otherwise, there will be some primitive self-coverage instrumentation conditionally inserted in the driver code that we can use at least for major code paths.

Cross-component testing plans

There's not much to do here apart from normal interactions with the ECM.

Upgrading/Supportability/Deployment design

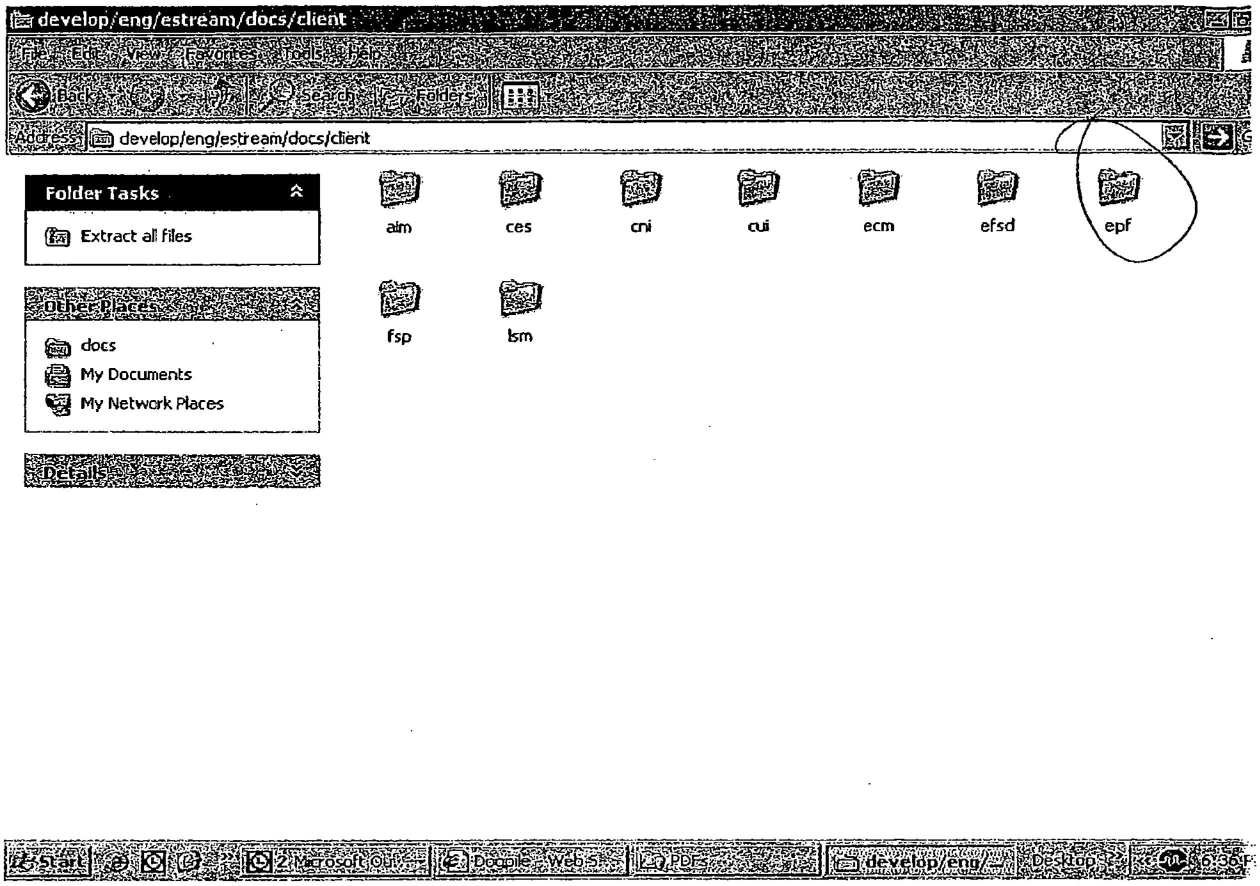
For supportability, there will be solid debugging aids—using printfs—built into the EFSD sources. Additionally, aside from good error codes returned from its interfaces, the EFSD will explore diagnostic traces optionally dumped for deployment.

Issues with stakes in the ground

- At present, I am assuming that there is a single drive letter associated with the EFSD, though there's no technical reason why this must be so. If indeed we organize the client system and the eStream file hierarchies to have multiple drive letters, either the EFSD or the ECM will need to parse the drive letter and do the right thing.
- This design assumes that 8.3 DOS-style filenames need not be supported. Adding such support will increase the complexity of the EFSD, as well as many other components in the eStream system: on the client, on various servers, and on the content builder.
- No support is provided in this design for:
    a. byte locks
    b. directory notification
    c. file open by file id
    d. file system control requests

Open Issues

1. I'm unclear on how to use the NT File Cache for metadata and directory contents. For now, I'm ignoring such matters, and we will only be caching file contents.
2. I do not know how to hook up the EFSD to UNC names. That is, I don't know how to set things up to have all file accesses like \\ASP1\Office\winword.exe be directed to the EFSD by the NT I/O Manager.
3. This design doesn't cover exactly how IRPs are posted for asynchronous processing. The SFSD example in Rajeev Nagar's book really isn't sufficient for some of what we need to do. Also, it's unclear to me what value there is to returning STATUS_PENDING and handling a request asynchronously if the caller is blocking anyway.



develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back
Search
Folders
Address
develop/eng/estream/docs/client
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start
develop/eng/
Desktop

[/STANDARD.01]

eStream 1.0 Prefetching & Fetching Low Level Design

*Anne Holler *[redacted]* Version 2.3*

Functionality

The EStream Prefetching and Fetching [EPF] module requests eStream application pages. The module handles demand fetching & prefetching. Prefetching is based on information in the prefetch section of the AppInstallBlock, on spatial locality, and potentially on profiling data gathered during previous runs of the application on this client.

Please see "eStream 1.0 Client Profile/Prefetch Straw Man" for background information related to prefetching & profiling. In particular, that document discusses the redundancy of gathering profiling information for prefetching on a client w/generous cache resources & the limitations of path-based prefetching when bandwidth is insufficient to hide latency without adequate execution-time lookahead. The collection & utilization of client profile information will be developed under an option (as it was for the prototype), and will be deployed in eStream 1.0 only if appropriate benefit is demonstrated.

Data type definitions

For core prefetching & fetching functionality, EPF uses the ECMRequest data structure.

For initial prefetch data, EPF reads FileNum,PageNum pairs from a file given by AIM.

For profiling data, EPF reads/writes FileNum,PageNum predecessor/successor per-AppID information from/to nonvolatile storage. This information is of a similar nature to the eStream cache and should be stored similarly on a per-client basis. For eStream 1.0, profile data is only stored if user has indicated that eStream cache can grow w/o a bound.

Interface definitions

From AIM to EPF:

- EPFPrefetchInitialAppBlocks(In AppID, In AppCorePrefetchSectionFileName, In AppOptionalPrefetchSectionFileName)

From ECM to EPF:

- EPFReportPage(In AppID, In FileNum, IN PageNum, In *Request)
- EPFReadPage(In AppID, In FileNum, In PageNum, In *Request)

From CUI to EPF:

- EPFPrefetchSet(In OnOff)

From EPF:

- LSMGetAccessToken(IN AppID, IN RequestHandle, IN ActivePrefetchOnly, IN RegisterCallback, OUT AccessTokenStatus, OUT AccessToken)
- ECMIsPageInCache(In AppID, In FileNum, In PageNum, In Priority, Out PageState)
- ECMReservePage(In AppID, In FileNum, In PageNum, In Priority)

Component design

Fetching and prefetching pages during program execution:

Interface with ECM [run in an ECM thread]:

```
ECM repeatedly calls EFSDGetRequest to pick up next eFSD request

EFSDGetRequest allocates ECMRequest data struct X & initializes

If the request is ERT_READ then

  ECM calls ECMReadPage w/ ECMRequest data struct X

  ECMReadPage acquires the work list/index lock

  If the READ hits in the cache then

    ECMReadPage calls EPFReportPage w/ AppId, FileNum, PageNum

    EPFReportPage allocates ECMRequest data structure Y,
       Puts the appropriate info including ERT_READ_HIT in it,
       Enqueues it in EPF's input queue & immediately returns

    ECMReadPage copies the data to the kernel and returns

  Else /* READ misses in the cache then */

    ECMReadPage calls EPFReadPage with AppID, FileNum, PageNum

    EPFReadPage allocates ECMRequest data structure Y,
       Puts the appropriate info including ERT_READ in it,
       Enqueues it in EPF's input queue & immediately returns

    /* EFSDCompleteRequest is called when data comes back from NW */

  End If /* READ hit or miss in cache */

  ECMReadPage releases the work list/index lock

Else if request is ERT_WRITE then

  /* For prototype, only prefetched & profiled based on READs */

End if /* Request is ERT_READ or ERT_WRITE */
```

EPF processing of its input queue [run in EPF thread]:

```
EPF repeatedly gets next ECMRequest in its input queue [using lock]:

Switch (RequestType)
  Case ERT_READ:
    Calls ECMReservePage with ECM Request data struct Y
    /* Fall through */
  Case ERT_READ_HIT:
    Call EPFRecordReference(In AppID, In FileNum, In PageNum)
    Call EPFPrefetchPredictedPath(In AppID, In FileNum, In PageNum)
    Break;
End switch
```

EPFPrefetchPredictedPath:

```
EPF predicts path based on current reference, on look-ahead distance,
  & potentially on previous references including stored profile data

For each page reference in the path

  EPF invokes ECMIsPageInCache

  If the page is not either in the cache or in flight then

    EPF calls ECMReservePage w/null ECMRequest (designates prefetch)

  End if

End for
```

EPFRecordReference:

```
EPF records pred/succ pairs of FileNum,PageNum on a per-AppID basis
```

Prefetching pages specified in AppInstallBlock:

```
  At application install time, LSM calls ECMMount & then AIMInstall

  AIMInstall calls EPFPrefetchInitialAppBlocks which returns immedi-
ately

  EPF uses separate prefetching thread to obtain this app's core pages:

    For each (FileNum,PageNum) pair from AppCorePrefetchSectionFileName

      EPF invokes ECMReservePage w/priority parameter marked as criti-
cal
        and w/null ECMRequest (designates prefetch)

  While cache utilization & client networking usage sufficiently low

    For each (FileNum,PageNum) from AppOptionalPrefetchSectionFileName
```

```
EPF invokes ECMReservePage w/null ECMRequest (designates pre-
fetch)
```

Prefetching pages independent of execution:

For client eStream installations with adequate caching resources, it may make sense to prefetch pages of files that are heavily-used & widely-covered. The winstone data indicated that such files include the main exe associated w/each application & certain core dlls. Hence, the idea of prefetching pages independent of eStream execution. This idea was not explored during the prototype phase & will be deployed in the eStream 1.0 product if it proves itself.

If the eStream client software is active for a particular user & it detects that the client system has been idle for an extended period of time, the independent prefetching system is engaged. The independent prefetching system decides which AppID would benefit from prefetching (somehow), which FileNum should have some of its pages prefetched (somehow), & invokes **LSMGetAccessToken** indicating ActivePrefetchOnly & specifying a callback routine to which LSM should report whether the AccessToken is obtained. ActivePrefetchOnly implies that the AccessToken should not be renewed automatically (to facilitate pre-emption of the AccessToken by a client who wants to use the app) and it implies that any problems getting the AccessToken should result in giving up rather than launching some dialog with the user. If the AccessToken is obtained, then as long as there are adequate cache resources & as long as the client network usage is low (aside from the bandwidth being used for prefetching!), EPF fetches pages in FileNum that are missing from the cache. The independent prefetching system disengages whenever client load detected, whenever an AccessToken is denied, whenever the cache is enhanced or polluted enough, etc.

Testing design

Unit testing plans/Coverage testing plans

A simple driver program, which will consist of a sequence of EPFReportPage & EPFReadPage calls and stubs for ECMIsPageInCache & ECMReservePage, will be used for unit testing of the runtime prefetching/fetching functionality. Expected profiling information will be compared with that collected and expected fetching/prefetching sequences will be compared with those produced. A dummy AppInstallBlock prefetch section file will be created and will drive unit testing of that portion of the prefetcher.

Stress testing plans

High fetching demand will be most stressful for this component. This may be tested by running many applications simultaneously on a client, forcing ECM to "miss" every reference, but having CNI return the data fairly quickly to keep the pressure on EPF.

Cross-component testing plans

This component is tightly coupled with the ECM. It will be integrated with the ECM as quickly as possible & will leverage off of ECM's testing infrastructure.

Upgrading/Supportability/Deployment design

A tracing feature will be introduced to have EPF report each fetch & prefetch it is making and why. An option to disable prefetching (and potentially profiling) will be provided.

eStream 1.0 Client Profile/Prefetch Straw Man

Anne Holler *[redacted]* *Version 1.0*

Introduction

This document presents background issues related to the collection of eStream client profile data & its use in client prefetching and client cache replacement policy and lists proposed design decisions wrt client profiling and prefetching for eStream 1.0. The document is intended to provide a baseline for discussions prior to proceeding with a detailed low level design.

Client Profile Data Background

eStream profile data may be used in client prefetching & client cache replacement policy. Ideally, eStream profile data consists of a sequence of (FileNum,PageNum) entries which comprise all references made on behalf of some run(s) of a particular eStream app, with the idea that any of these references would potentially result in an eFS request. Dynamic page tracing based on page faults gives a subset of the ideal trace, which includes the page references for compulsory misses, some subset [possibly empty] of the page references for the non-compulsory misses, and no entries for the remaining page references. A recently proposed augmentation of this data with information from static analysis of executables misses indirect links between pages and is in general imperfect; subprogram entry points will not always be visible and code & data may be interleaved on the same page in x86 code. Gathering ideal eStream profile data is a heavy-weight process, involving instruction level tracing.

What kinds of issues could arise with respect to incomplete eStream profile data? To illustrate, let us assume that the sequence {b,c,d,x,y,z} appears in an incomplete eStream profile as {b,z}. One issue is that we do not know how far ahead a page in an incomplete profile will actually be used; prefetching z at the time b is being accessed is a problem if z displaces any of {c,d,x,y}. Another issue is that we may miss prefetch opportunities; the incomplete profile does not indicate that we need to prefetch {c,d,x,y} if accesses to them page fault and the pages are not in our cache. Another issue is that incomplete profile data does not allow us to know which pages are used frequently; if b is used frequently, we might see only one reference to it per run in the ordered profile data (because its frame tends to stay in main memory), whereas if z is used infrequently, we might see no references to it on some runs and possibly multiple references to it on other runs (assuming its infrequent use means its frame tends to get displaced from physical memory). Hence, straightforward use of either reference counts or LRU based on incomplete eStream profile data can be flawed wrt client cache replacement.

If incomplete profile data based on page faults is both collected and used on the same client, the issues cited above are less likely to occur (assuming the execution load on the client is roughly the same each time the application is invoked); profile data collected on an eStream builder machine and downloaded to a customer client machine at install time would be expected to be more likely to encounter these issues. While various approaches have been discussed to mitigate the potential impact of these problems, the decision has been made to forgo the profile section of the AppInstallBlock for eStream 1.0 (largely because the planned lightweight builder training strategy means that the profile section contents would be redundant with the prefetch section contents). Uploading client profile data, which has previously been discussed, is also outplan for eStream 1.0; since there is no initial downloaded profile data, there is no need to use uploaded client profile data to refine it. Application usage data, which may be of interest to ISVs and others, will be gathered at a less granular level from the SLM server.

What good is page-fault-based client profile data collected & used on the same client? If the client has a cache of unlimited size, in which no block is ever replaced, this profile data is redundant in the sense that it brings no obvious value to eStream 1.0 in addition to what is already provided by the contents of the client cache. If the client cache does need to do replacement, this profile data can be used to guide the replacement strategy, but only if the data is used in a clever way to avoid the oddities that result from its incompleteness (e.g., use per run); if it is not used cleverly, one may as well use random replacement. Client cache replacement implies that refetching of displaced pages may be needed. The profile data can potentially do a good job of supporting prefetching of this refetching; in experiments with the prototype, one can delete a warmed client eStream cache, just use the profile data, & hide virtually all latency associated with the refetches, at least in the scenario in which there is adequate bandwidth. For a 100Mbps link, executing start/exit of word, cache hit rate with profile-guided prefetching is 98%, with 6% additional overhead for redundant prefetch.

How does profile-based prefetching compare with simple locality-based prefetching, for which no special technology is needed to collect and use the profiles is needed? In experiments with the prototype, locality-based prefetching yielded not-too-terrible results as compared with profile-based prefetching wrt client cache hit rate, but at the cost of a much higher wasted prefetch rate, i.e., at the cost of much more network bandwidth and server traffic. For 100Mbps link, executing start/exit of word, cache hit rate with locality-guided prefetching is 75%, with 89% additional overhead from redundant prefetch. We need to decide if client profiling is worthwhile for eStream 1.0, based on whether we believe there will be sufficient cache replacement and if so, whether we want to avoid the extra overhead of locality prefetching to restore displaced pages and if there will be sufficient bandwidth to exploit the extra knowledge that profile data gives us [perhaps via idle prefetch]. Putting a stake in the ground, I advocate designing and building client profile gathering and associated prefetching, because I believe it may be useful and because I believe we will be able to learn things from it which may profit us in the future. Please note that for the prototype, client profile gathering was under an #ifdef, so it could be removed as desired, since it adds memory consumption and overhead to the eStream client. We could develop this way for the real product as well, in case we decide

that it is not worthwhile for the eStream 1.0 product at all or at least for some particular variant of the product which is geared to very fat cache clients.

Please note that, in general, client profiling is done on a per application basis; if the user is running several eStream applications on his client, it is expected that we would want to learn page sequences within each application, not sequence relationships across apps. Similarly, client profiling is done on a per invocation basis; if the user is running several instances of a particular eStream application, it is expected that we would want to learn about the sequences belonging to each instance separately. For the prototype, we turned off learning profile sequences if more than one instance of an application was running; we may want to consider doing this for the real product as well.

Client Prefetching Background

When does a client do eStream prefetching? The most intuitive time to do eStream client prefetching is at eStream app runtime, in association with runtime behavior; this is the only time at which prefetching is done in the prototype. Two other times to consider doing eStream prefetching are at app installation time and at network idle time. At app installation time, the AppInstallBlock contains a prefetch section, which specifies {FileNum,PageNum} pairs for pages which are recommended to prewarm the client cache for this app; the client prefetcher will ask the client networking code to request these pages from the appropriate app server. At network idle time (i.e., when there is little client networking traffic), app pages may be downloaded without directly impacting client eStream performance; clearly such prefetching would need to be done judiciously to avoid polluting the client's cache, impacting server scaling, wasting network bandwidth, or hogging the app's license, but this could be a interesting strategy for eStream technology if it were done effectively.

What does the eStream client prefetch? As discussed in the previous section on client profiling, profiling info can be used to drive which pages to prefetch, either by prefetching along an observed path or by prefetching according to a projected pattern, possibly based on observed references. Spatial locality is a frequently observed pattern in general, and may be used as the default projected pattern in the absence of relevant profiling information. In eStream 1.0, we may also consider using semantic information to drive prefetching. One example might be prefetching the contents of the directory files for an installed app, since directory files may contain useful FileIDs and metadata. Another example might be to prefetch any missing pages of heavily used files such as the main executable for each application, with the idea that the pages might eventually be used for some app feature not yet exercised. Several folks have suggested that we gradually download the entire app, if cache space allows, with the vision that this provides full native performance; if this can be done without impacting any of our core selling points, this might be worth considering.

Client Profile Data Collection Design Proposal

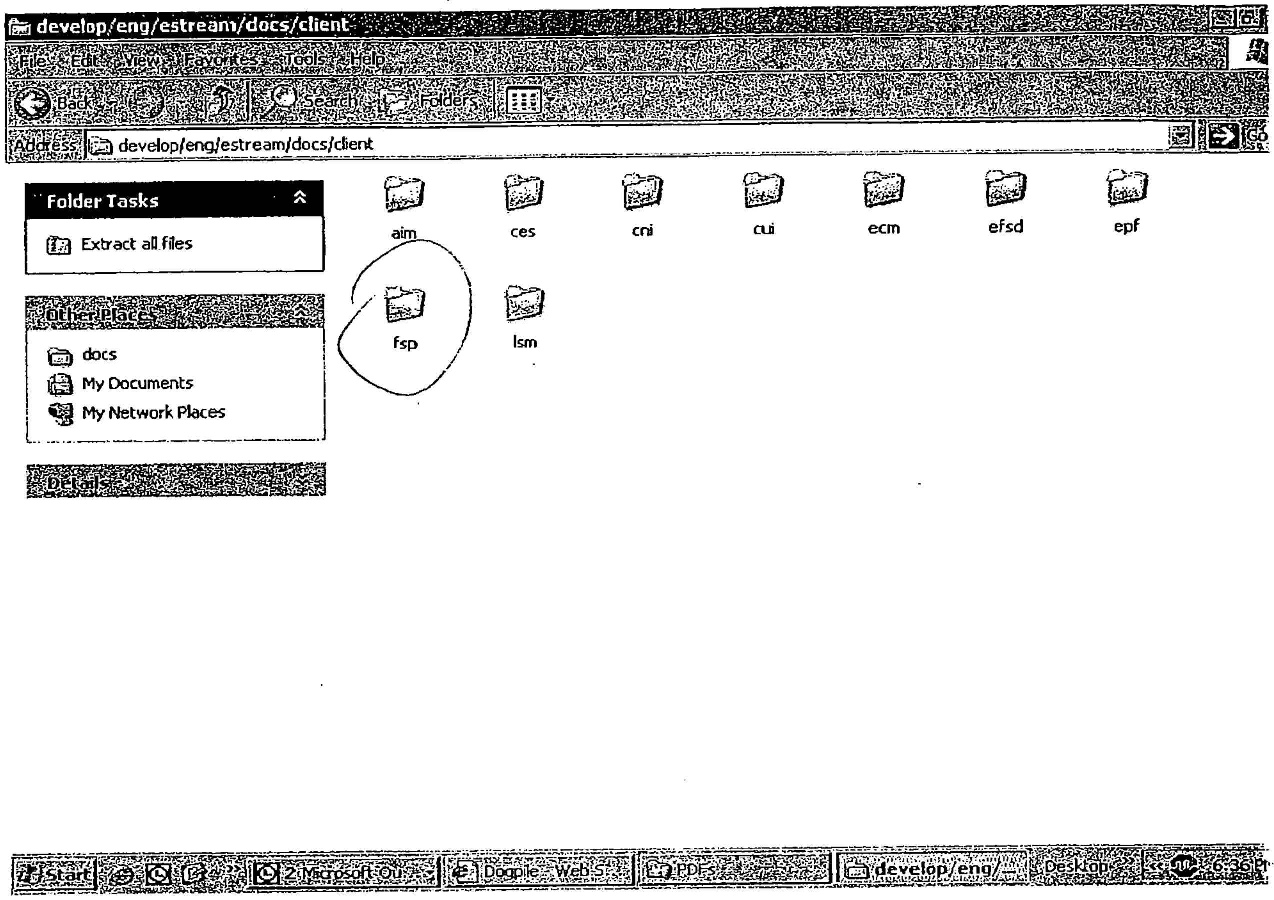
Here are key points on the design of eStream 1.0 client profile data collection:

*) Have profiler see all requests coming from eFSD
*) Record profile data on per-application basis
*) Record profile sequence for app only if single instance of app running (how can tell?)
*) Represent (potentially lengthy) profile data list (with some loss) as (pred,succ) tuples
*) Make client profiler optional
*) Mark use-per-run pages high priority to aid cache replacement policy

Client Prefetching Design Proposal

Here are key points on the design of eStream 1.0 client prefetching:

*) Prefetch at app install, mark specified pages as high priority to avoid their replacement
*) Also prefetch at app runtime, have prefetcher see all requests coming from eFSD
*) If eFSD page request on known path, project next refs on path; else, use spatial locality
*) Prefetch only projected refs that are not already in client cache
*) Consider network speed, path likelihood, cache issues in determining prefetch distance
*) Also prefetch at idle, if unlimited cache; get missing pages from app's main executable


develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back
Search
Folders
Address develop/eng/estream/docs/client
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start
2 Microsoft Ou
Dogpile Web S
PDFs
develop/eng/...
Desktop
6:36

eStream File Spoofer Low Level Design

Curt Wohlgemuth
Version 2.0
[redacted]

Functionality

The eStream file spoofer is a kernel-mode driver responsible for redirecting file accesses **from** local file systems **to** the eStream file system driver. It is implemented as a file system filter driver that traps all IRP requests to the file system device handling drives that must be spoofed, and redirects these requests to the EFSD.

Data type definitions

The file spoofer will understand entries in the "file spoof database" as they have been identified by the eStream builder and installed by the app install manager, but these are not defined by this component.

Entries in this spoof database will have the following entries:

- original (fully qualified) path name of file: this resides somewhere on a local disk of the client machine
- new (fully qualified) path name of the file to spoof to: this resides on the eStream file system drive

The spoof database will reside in the registry, so it can be persistent across reboots, and so the file spoofer need not open a file to load it. As applications are installed on a client machine, the Application Install Manager will load new spoof entries into the registry, and inform the file spoofer that it must reload this database. Similarly, when an app is uninstalled, the AIM will remove spoof entries from the registry, and inform the file spoofer.

This proposes that the each spoof entry is a separate name/value entry under a single key in the registry:

- Name: the original filename
- Value: the new filename

Interface definitions

The eStream file spoofer is called by two components:

1. The eStream client start service, which will start and stop the file spoofer

2. The AIM, which will inform the file spoofer to reload the spoof database from the registry

The interfaces called by both of these user-mode components will be in the form of DeviceIoControl() calls. The following IOCTL codes will be defined for use by callers of the file spoofer:

```
IOCTL_FS_START_SPOOFING
IOCTL_FS_STOP_SPOOFING
IOCTL_FS_RELOAD_SPOOF_DB
```

Starting spoofing

The input buffer for this IOCTL should supply the name of the registry key containing the spoof entries as values. The output buffer for this IOCTL is ignored and should be NULL.

This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to read the spoof registry entries, and load up each entry into memory.

Note that starting spoofing is currently identical to reloading the spoof database.

This is called by the eStream client start service on startup.

Stopping spoofing

The input and output buffers for this IOCTL are ignored and should be NULL. This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to clear its memory image of the spoof database.

This is called by the eStream client start service on shutdown.

Reload spoof database

The input buffer for this IOCTL should supply the name of the registry key containing the spoof entries as values. The output buffer for this IOCTL is ignored and should be NULL.

This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to read the spoof registry entries, and load up each entry into memory.

Note that this is currently identical to starting the spoof database.

This is called by the AIM when a new eStream app is installed.

Component design

The file spoofer will have these major tasks:

- Track the following data:
  - all current valid file spoof entries
  - spoof entries by filtered file system
- Filter native file system drivers for local drives, intercept all IRP_MJ_CREATE and FAST_IO_QUERY_OPEN requests, and for spoofed files, change the file-name of the FileObject associated with these requests.

Data structures

The file spoofer needs to be able to quickly look up a filename in the in-memory spoof database. The current design will use a hash table, whose size and hash function will be tuned as we get experience with real applications.

Adding or deleting entries from the hash table will by synchronized using a global resource.

Algorithms

Here are basic algorithms for these steps.

Load spoof database

This reads all the name/value pairs under the registry key which holds the spoof entries, loads them into a temporary hash table, then points the real hash table to this one.

```
traverse the registry until the input registry key is opened, using ZwOpenKey()
if not found
    return error
if no name/value pairs exist in this key
    return "no data"

for each name/value pair found with ZwEnumerateValueKey()
    build a hash node for this and insert it into temp hash table
    if the drive letter for the old filename entry is one we're not currently filtering
        put this drive letter on new drive list

acquire the hash table resource exclusively
point the global hash table head pointer to the temp hash table
release the resource

for each drive letter on the new drive list
```

```
    look up FS device for this drive
    if we really aren't attached to it
        attach self to this FS device as filter driver

free the old hash table
free the drive list
return success
```

Stop spoofing

```
acquire hash table resource exclusively
free the global hash table
detach self from all filtered FS devices
release hash table resource
```

Trap Create and QueryOpen requests

```
acquire the hash table resource shared
if hash table head pointer is non-NULL
    look up input filename in hash table
release hash table resource
if filename not found in hash table
    send I/O request to original target FS driver
else
    free memory of existing file name in input FileObject
    allocate memory for new, spoofed filename, copy into this memory
    send I/O request to eStream file system driver
```

Testing design

Unit testing plans

The file spoofer will be tested as a standalone component, apart from the rest of the eStream client. A driver test program will be written to test all functionality and corner cases. This includes filtering all FSDs active for a client system, and multiple drives handled by a single FSD.

Stress testing plans

The file spoofer should be able to work, with little or no performance cost to the system as a whole, even when the attached FSDs are under heavy load. Some stress testing will be done in this fashion.

Coverage testing plans

If we come up with a method for measuring coverage for kernel-mode components, we'll do so for the file spoofer as well.

Cross-component testing plans

Not clear if anything need be done here outside of the standard execution of the eStream client.

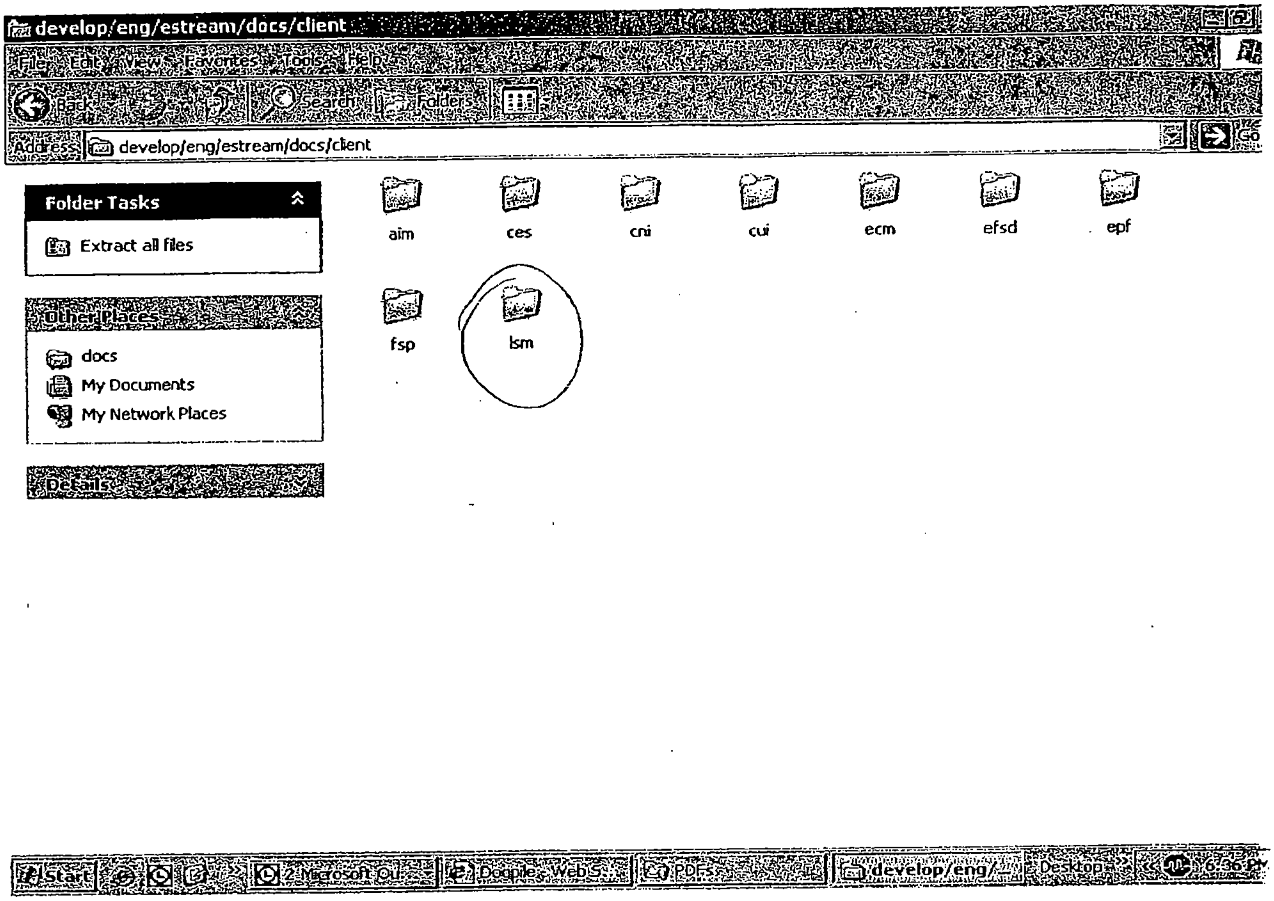
Upgrading/Supportability/Deployment design

I don't see any upgrade/compatibility issues for the file spoofer. For supportability, there will be a good debugging strategy and sufficient error message return codes for the caller.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

1. We will need to experiment with the hash table to tune it for fast lookup. It's possible that we may need to replace the hash table with a faster lookup algorithm.


develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back
Search
Folders
Address
develop/eng/estream/docs/client
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start
Dogpile
PDFs
develop/eng/...
Desktop

[/STANDARD.01]

eStream 1.0 Client License Subscription Mngr Low Level Design

Anne Holler * [redacted] * *Version 2.9*

Functionality

This document presents the low level design of the License Subscription Manager [LSM], an eStream client module that tracks information about ASP accounts, application subscriptions, and installed applications. The document also describes the interface to the eStream Client Browser Module [CBM], a client component that allows an ASP web server to notify LSM of client-relevant changes in an ASP account. (The CBM interface is used in specific situations, as described below; ASP account changes are detected automatically in steady-state operation.)

Data definitions

There are two data sets that LSM maintains on a per-user basis on the client [*italics* => only volatile storage]. One set is associated with the ASPs from which the client eStreams applications & the other is associated with the eStream applications to which the client is subscribed. This data is kept in the registry.

ASP Data Set

- uInt128 AspID -- Local handle for client convenience
- string UserName
- *string Password* – Only volatile storage unless user approves nonvolatile
- string ASPWebServerName -- May simply be an ip address
- eStreamServerSet SLiMServerSet

Subscription Data Set

- uInt128 SubscriptionID
- uInt128 AppID
- Int32 RootFileNumber
- *eStreamServerSet AppServerSet*
- *eStreamAccessToken AccessToken*
- *uInt64 ExpectedExpirationTime*
- enum {Installed, NotInstalledPrompt, NotInstalledDontPrompt} AppInstallStatus
- enum {Updated,NotUpdatedPrompt,NotUpdatedDontPrompt} AppUpgradeStatus
- enum {ActiveInUse, ActivePrefetchOnly, Inactive} AppActiveStatus
- string AppName
- string VersionName
- string Message
- uInt128 AspID -- Local handle

Interface definitions

Client Component Acronyms

- AIM: Application Installation Manager
- CBM: Client Browser Module
- CES: Client EStream Startup
- CNI: Client Network Interface
- CUI: Client User Interface
- ECM: EStream Cache Manager
- LSM: License Subscription Manager

LSM InterfacesTo/From Server Components

From LSM to SLiMServer via CNI:

- CNIGetSubscriptionList(IN AspID, IN UserName, IN Password)
- CNIGetLatestApplicationInfo(IN AspID, IN SubscriptionID)
- CNIAcquireAccessToken(IN AspID, IN SubscriptionID, IN UserName, IN Password)
- CNIRenewAccessToken(IN AspID, IN SubscriptionID, IN UserName, IN Password, IN AccessToken)
- CNIReleaseAccessToken(IN AspID, IN SubscriptionID, IN UserName, IN Password, IN AccessToken)
- CNIRefreshAppServerSet(IN AspID, IN SubscriptionID IN AccessToken, IN BadQos, IN NoService)

From ASPWebServer to CBM:

- ASPNotify(IN UserName, IN SLiMServerSet)

From LSM to ASPWebServer:

- PromptASPNotify: HTTP POST to server cgi script to engender an ASPNotify

LSM Interfaces To/From Client Components

From ECM to LSM:

- **LSMGetAccessToken**(IN AppID, IN RequestHandle, IN ActiveInUse, IN RegisterCallback, OUT AccessTokenStatus, OUT AccessToken)
- **LSMReleaseToken**(IN AppID)

From LSM to ECM:

- ECMMount(IN AppID, IN RootFileNumber)
- ECMRemount(IN AppID, IN OldRootFileNumber, IN NewRootFileNumber)
- ECMSetTokenState(IN AppID, IN State) – *NotHolding*, *Holding*, *Denied*

From CBM to LSM:

- **LSMAspAccntPing**(IN AspWebServerName, IN UserName, IN SLiMServerSet)

From CES to LSM:

- **LSMInitialize**(IN WindowsUserName) -- Read ASP & Subscription Data Sets from registry
- **LSMUpdateAllSubscriptionStatus**(void)

From LSM to AIM:

- AIMInstall(IN AppID, IN AppInstallBlockFileName, OUT InstallStatus)
- AIMUninstall(IN AppID, OUT UninstallStatus)

From LSM to CUI:

- CUIInformUser(IN Message)
- CUIAskUserYesNo(IN Message, IN CheckAskAgain, OUT Response, OUT DontAskAgain)
- CUIAskUserPassword(IN Message, OUT Response, OUT Retain)

From CUI to LSM:

- **LSMUpdateAllSubscriptionStatus**(void)
- **LSMGetAspList**(OUT NumAsps, OUT AspID[])
- **LSMGetAspInfo**(IN AspID, OUT ASPWebServerName, OUT UserName)
- **LSMGetAppList**(IN AspID, OUT NumApps, OUT AppID[])
- **LSMGetAppInfo**(IN AppID, OUT AppName, OUT VersionName, OUT Message, OUT AppInstallStatus, OUT AppUpdateStatus)
- **LSMInstall**(IN AppID, OUT InstallStatus)
- **LSMUninstall**(IN AppID, OUT UninstallStatus)
- **LSMUpgrade**(IN AppID, OUT UpgradeStatus)

From EPF to LSM:

- **LSMGetAccessToken**(IN AppID, IN RequestHandle, IN ActivePrefetchOnly, IN RegisterCallback, OUT AccessTokenStatus, OUT AccessToken)

From CNI to LSM:

- **LSMGetAppServerSet**(IN AppID, OUT AppServerSet)
- **LSMGetNewAppServerSet**(IN AppID, OUT AppServerSet)
- **LSMGetSLiMServerSet**(IN AspID, OUT SLiMServerSet)
- **LSMGetNewSLiMServerSet**(IN AspID, OUT SLiMServerSet)
- **LSMGetAccessToken**(IN AppID, IN RequestHandle, IN Unknown, IN RegisterCallback, OUT AccessTokenStatus, OUT AccessToken)
- **LSMReturnSubscriptionList**(IN AspID, IN ReturnCodes, IN NumberOfSubscriptions, IN SubscriptionID[])
- **LSMReturnLatestApplicationInfo**(IN AspID, IN SubscriptionID, IN ReturnCodes, IN UpgradeInfo)
- **LSMReturnAcquireAccessToken**(IN AspID, IN SubscriptionID, IN ReturnCodes, IN AccessToken, IN RenewalFreq, IN AppServerSet, IN UpgradeInfo)
- **LSMReturnRenewAccessToken**(IN AspID, IN SubscriptionID, IN ReturnCodes, IN AccessToken, IN RenewalFreq, IN AppServerSet)
- **LSMReturnReleaseAccessToken**(IN AspID, IN SubscriptionID, IN ReturnCodes)
- **LSMReturnRefreshAppServerSet**(IN AspID, IN SubscriptionID, IN ReturnCodes, IN ServerSet)

Component design

The LSM is a service module that handles requests from eStream client software and from the ASPWebServer via the CBM [discussed under its own heading at the end of this section]. Hence, LSM's component design is presented here in terms of how it responds to the requests that it receives in the three main scenarios in which it participates: running an eStream application, getting/refreshing application subscription information, and displaying subscription information.

The web pages at \\fserv2\home\webpages\pmain.html are intended to provide a framework for discussing overall issues related to subscribe/install/uninstall scenarios. The discussion of scenarios in this section is at a much lower level than that material.

Run an Application

- ECM invokes **LSMGetAccessToken** whenever there is a reference to a file belonging to a particular AppID for which ECM's internal data structure indicates that the AccessToken is in the *NotHolding* state. [ECM can determine the AppID of an eStream app by inspecting the fully-specified eFSD filename; the first level of the directory path is a string version of the AppID. If LSM does not recognize this AppID for some reason, an appropriate status is returned to ECM.]
- LSM marks the AppID as ActiveInUse in the Subscription Data Set & returns the appropriate status to ECM. If the status indicates that LSM has an AccessToken for the application, ECM updates its internal data structure to indicate the AccessToken is in the *Holding* state & continues. If the status indicates that LSM

does not have an AccessToken for the application, LSM takes ownership of the specified request, holding the RequestHandle in an internal queue for an eventual reissue request to eFSD.

- LSM issues a **CNIAcquireAccessToken** call via CNI to a SLiMServer & when LSM receives the AccessToken via the **LSMReturnAcquireAccessToken** call-back, it places a copy of it in the volatile Subscription Data Set, assigning ExpectedExpirationTime according to the RenewalFrequency indicated by the SLiMServer and setting up a timer alarm to go off shortly before the AccessToken is expected to expire. It calls **ECMSetTokenState** to notify ECM that the AccessToken is in the *Holding* state for the AppID.
- LSM compares the application's current RootFileNumber on the client with UpgradeInfo.RootFileNumber returned wrt the **CNIAcquireAccessToken** call; if the latter is higher & UpgradeInfo.ForcedUpgrade is off, it calls **CUIAskUserYesNo** about upgrading. LSM sets AppUpgradeStatus and RootFileNumber fields in the Subscription Data Set appropriately & calls **ECMRemount** if there is a minor upgrade.
- Should an AccessToken be denied due to a mandatory major upgrade that costs money, LSM calls **CUIInformUser** to let him know that he needs to interact with his ASP to revise his account.
- Should an AccessToken be denied due to a mandatory major upgrade that does not cost money, LSM calls **CUIAskUserYesNo** about upgrading. If he responds affirmatively, the Subscription Data Set is updated, **AIMUninstall** is called to nuke the current version, & **AIMInstall** is called to install the upgraded version.
- Should an AccessToken be denied because it is currently held by another of the user's clients, a copy of the other client's AccessToken is returned. If the user responds affirmatively to **CUIAskUserYesNo("Yank?")**, then LSM will call **CUIInformUser("Please wait")**, send the token to **CNIReleaseAccessToken**, & then retry **CNIAcquireAccessToken** at the appropriate time.
- For AccessToken denials that stick [either because the user has decided not to take a mandatory upgrade at this time, or because the user has decided not to yank another of his clients' AccessToken, or because the user has some problem with his ASP account (didn't pay, account suspended/terminated, got evicted, authorization failed)], the user is informed of the problem and told to save/exit the application ASAP. **ECMSetTokenState** is called to mark the token state *Denied* and EFSD is told to retry the request. When ECM's internal data structure has the AccessToken status for an application marked *Denied*, it allows the user to continue running out of the cache, but will report failure to EFSD immediately on any access that misses in the cache.
- CNI frequently invokes **LSMGetSLiMServerSet** to obtain SLiMServers for AccessToken requests. Should CNI have quality of service issues with those servers, it calls **LSMGetNewSLiMServerSet**; LSM uses the **PromptASPNotify** interface to engender the ASP Web Server to send a message containing a SLiMServerSet.
- Whenever an AccessToken timer alarm goes off, LSM checks that the associated app is marked ActiveInUse & if so, it issues a **CNIRenewAccessToken** & puts token returned by **LSMReturnRenewAccessToken** in the Subscription Data Set, resetting ExpectedExpirationTime and scheduling another timer alarm.

- When CNI needs to fetch requested data from an AppServer, it invokes **LSMGetAccessToken.**
- CNI frequently calls **LSMGetAppServerSet** to obtain a set of AppServers through which to cycle; LSM returns the servers associated with the most recently acquired AccessToken. Should CNI be dissatisfied with the quality of service associated with the AppServers it is given, it calls **LSMGetNewAppServerSet** which sends **CNIRefreshAppServerSet** to a SliMServer and receives a **LSMReturnRefreshAppServerSet** callback.
- Should a CNI request be denied by an AppServer because the submitted AccessToken was expired, CNI calls **LSMGetAccessToken**, which operates as described above with respect to acquiring the AccessToken and with respect to sending a reissue request to eFSD.
- ECM invokes **LSMReleaseToken** whenever there is a transition from 1 to 0 open files wrt a particular eStream app, setting the AccessToken status in its internal data structure to *NotHolding* (from either *Holding* or *Denied*). LSM sends a **CNIReleaseAccessToken** call to a SLiMServer [possibly only if the ExpectedExpirationTime for the AppID in question is not in the past] & receives an **LSMReturnReleaseAccessToken** callback. LSM marks the application as Inactive in the Subscription Data Set, indicating that automatic AccessToken renewal should not be continued for this application [LSM may also cancel any outstanding alarm].

Get/Refresh Subscription Information

- Whenever the CBM receives an **ASPNotify** from an ASPWebServer, it sends an **LSMAspAccntPing** message to the LSM. The LSM checks its ASP Data Set to see if the specified ASP is one that it recognizes. If LSM does not recognize the ASP, it allocates a new ASP Data Set entry and calls **CUIAskUserPassword** to prompt the user for the password associated with this ASP+UserName (asking the user if the password can be saved on the client for future use). If LSM does recognize the ASP, it only calls **CUIAskUserPassword** if the user did not allow the password to be previously recorded.
- If the user indicates that his password can be saved on the client for future use, we encrypt the password using a reasonably secure symmetric algorithm, such as DES, the key being a hash of the AspID and the username. If we implement the eStreamClientExecutable as a service (outstanding implementation issue in CUI LLD), we store the password in a registry key that cannot be read by ordinary user accounts (services can run in the context of a separate account), else we store the password in a registry key belonging to the user.
- Once LSM has the UserName & Password, it issues a **CNIGetSubscriptionList** request to one of the ASP's SLiMServers. For any SubscriptionIDs associated with that ASP that are installed on the client but do not appear in the list returned by the **LSMReturnSubscriptionList** callback, the user is informed that he cannot run the app at the current time & is asked if he would like it uninstalled from his client (note that we may be legally required to remove it from the cache, so this

may not be a question but rather just a message); **AIMUninstall** is invoked to achieve this.

- For all other SubscriptionIDs, a **CNIGetLatestApplicationInfo** request is sent to a SliMServer & a **LSMReturnLatestApplicationInfo** callback provides the data.
- For SubscriptionIDs already in the Subscription Data Set, the RootFileNumber is compared to see if the application has been upgraded. If it has, the user is informed; the new RootFileNumber is recorded (assuming either that the upgrade is mandatory or that the user indicated he wanted it) and **ECMRemount** is called to notify ECM about the upgrade so that it can take appropriate action. [Please note that for eStream 1.0, we will not support minor upgrades that involve new AppInstallBlocks; if a new AppInstallBlock is needed, the upgrade is by definition a major upgrade & the app needs to have a new AppID assigned & uninstall/install actions taken.]
- For SubscriptionIDs that were not already in the Subscription Data Set, a new Subscription Data Set entry is created and **CUIAskUserYesNo**("Install App?") is invoked. If user responds in the affirmative, LSM requests an AccessToken, calls **ECMMount** to notify ECM to prepare to service requests on behalf of this new application & then invokes **AIMInstall**, which can use the eFSD/ECM path to obtain the AppInstallBlock. If user responds in the negative, he is able to indicate "don't ask again"/"ask again later", which is stored in the Subscription Data Set.
- Whenever the eStream client software starts up or at specific times requested by the user via the ClientUI, **LSMUpdateAllSubscriptionStatus** is called to refresh the eStream client's knowledge of the apps to which the user is subscribed.

Display Subscription Information

- The CUI provides a user interface to display ASP and application info, using **LSMGetAspList, LSMGetAspInfo, LSMGetAppList, LSMGetAppInfo**.
- Please see "eStream 1.0 Client User Interface Low Level Design" for information on **LSMInstall, LSMUninstall**, and **LSMUpgrade.**

Client Browser Module Overview

The CBM is an eStream client component that runs in association with the client browser. CBM communicates a small amount of data to the core eStream client software (as described above) and it is realizable on both Internet Explorer 4+ and Netscape Navigator 4+. It does not have to start the eStreamClientExecutable if it is not already running.

Testing design

Unit testing plans/Coverage testing plans

For unit testing of LSM, a simple driver program will be written which will simulate the three main scenarios highlighted in this design document. It will exercise all LSM external entry points (listed in bold in the Interface Definitions section of this document) and

will achieve ~100% PFA coverage of all non-assert paths as measured by Rational Pure-Cov. A process will be introduced to automate the execution of this testing.

Cross-component testing plans

As defined in the high-level design, LSM has many cross-component interfaces. In the steady-state operation of eStream, its most heavily-used interfaces will be those with ECM and CNI. It is anticipated that LSM will be integrated first with these two pieces and will be added to an automated nightly testing of running eStream applications.

At eStream client software activation time, the CES interface is exercised; integration with CES and automated testing of the pair should be deployed as soon as both are available. At application install and uninstall time, the AIM interface is utilized; presumably, automated testing of installation & uninstallation of apps will be key to the AIM testing strategy and can be combined with the testing of this module when the two are integrated.

The CUI has a number of interfaces to LSM. Automated testing of the pair will be based on a UI testing tool like Rational's Visual Test tool (which BTW drives Winstone).

Stress testing plans

The most stressful usage scenario for LSM is high client load, i.e., many eStream applications running simultaneously. Visual Test will be used to set up this kind of testing scenario.

Upgrading/Supportability/Deployment design

LSM is heavily involved in user scenarios involving upgrading, supporting, and deploying eStream applications. The normal operation of these scenarios is automatic and seamless, but all are intended to have CUI interfaces that will allow manual intervention and troubleshooting.

In addition, LSM will provide a debugging interface for dumping the in-memory images of the ASP Data Set and the Subscription Data Set.

Implementation Issues

- Develop mechanism for implementing CBM. The plan is to make CBM a browser helper application, i.e., a standard windows32 app that is registered to handle a particular MIME encoded filetype (*.estream) in the HTML stream. [BTW, this is done by SoftwareWow.] As an alternative proposal, Bhaven created JavaScript that starts Excel & communicates with it, though this solution seems to require that the browser be Internet Explorer, that the app be ActiveX, and that the security setting on the browser be at minimum setting.

- LSM may move to kernel space and its implementation will avoid constructs which might unduly complicate this potential migration.

eStream 1.0 Client License Subscription Manager Straw Man

*Anne Holler * [redacted] * Version 1.2*

Introduction

This document presents background information related to the License Subscription Manager [LSM], an eStream client module that tracks information about ASP accounts, application subscriptions, and installed applications. The document also describes the interface to the eStream Client Browser Module [CBM], a client component that allows an ASP web server to notify LSM of client-relevant changes in an ASP account. (The CBM interface is used in specific situations, as described below; ASP account changes are detected automatically in steady-state operation). The document is intended to provide a baseline for discussions prior to proceeding with a detailed low-level design.

LSM Data Sets

There are two data sets that LSM maintains on the client in nonvolatile memory. One set is associated with the ASPs from which the client eStreams applications and the other is associated with the eStream applications to which the client is subscribed.

ASP Data Set

uInt128 AspID -- Local handle for client convenience
string UserName
string Password
string ASPWebServerName
eStreamServerSet SLiMServerSet

Subscription Data Set

uInt128 SubscriptionID
uInt128 AppID
Int32 RootFileNumber
eStreamServerSet AppServerSet
eStreamAccessToken AccessToken
uInt64 ExpectedExpirationTime
enum {Installed, NotInstalledPrompt, NotInstalledDontPrompt} AppInstallStatus
enum {ActiveInUse, ActivePrefetchOnly, Inactive} AppActiveStatus
string AppName
string VersionName
string Message
uInt128 AspID -- Local handle

LSM Interfaces To/From Client Components

The use of the interfaces listed below is discussed in the **LSM Functionality Overview** section of this document; the interfaces are summarized here for easy access.

Client Component Acronyms
AIM: Application Installation Manager
CBM: Client Browser Module
CES: Client EStream Startup
CNI: Client Network Interface
CUI: Client User Interface
ECM: EStream Cache Manager
LSM: License Subscription Manager

From ECM to LSM:
LSMNotifyAppStart(IN AppID, OUT AppStatus, OUT RootFileNumber)
LSMNotifyAppStop(IN AppID)
LSMCheckAccessToken(IN AppID, OUT AccessTokenStatus)
LSMGetAccessToken(IN AppID, OUT AccessToken)

From LSM to ECM:
ECMMount(IN AppID, IN RootFileNumber)

From CBM to LSM:
LSMAspAccountPing(IN AspWebServerName, IN UserName, IN SLiMServerSet)

From CES to LSM:
LSMUpdateAllSubscriptionStatus(void)

From LSM to AIM:
AIMInstall(IN AppID, IN AppInstallBlockFileName, OUT InstallStatus)
AIMUninstall(IN AppID, OUT UninstallStatus)

From CUI to LSM:
LSMUpdateAllSubscriptionStatus(void)
LSMGetAspList(OUT NumAsps, OUT AspID[])
LSMGetAspInfo(In AspID, OUT ...)
LSMGetAppList(In AspID, OUT NumApps, OUT AppID[])
LSMGetAppInfo(In AppID, OUT ...)

From LSM to CUI:
AskUserYesNo(IN Message, OUT Response)
AskUserPassword(IN Message, OUT Response, OUT Retain)
InformUser(IN Message)

From CNI to LSM:

```
LSMGetAppServerSet(IN AppID, OUT AppServerSet)
LSMGetNewAppServerSet(IN AppID, OUT AppServerSet)
LSMGetSLiMServerSet(IN AppID, OUT SLiMServerSet)
LSMGetNewSLiMServerSet(IN AppID, OUT SLiMServerSet)
```

LSM InterfacesTo/From Server Components

From ASPWebServer to CBM:

```
ASPNotify(IN UserName, IN SLiMServerSet)
```

From LSM to ASPWebServer:

PromptASPNotify: HTTP POST to server cgi script to engender an ASPNotify

From LSM to SLiMServer via CNI:

```
GetSubscriptionList(IN UserName, IN Password,
                    OUT ReturnCodes, OUT NumberOfSubscriptions,
                    OUT SubscriptionID[])
GetLatestApplicationInfo(IN SubscriptionID,
                         OUT ReturnCodes, OUT UpgradeInfo)
AcquireAccessToken(IN SubscriptionID, IN RootFileNumber,
                   IN UserName, IN Password,
                   OUT ReturnCodes, OUT AccessToken, OUT RenewalFreq,
                   OUT AppServerSet, OUT UpgradeInfo)
RenewAccessToken(IN AccessToken, IN UserName, IN Password,
                 OUT ReturnCodes, OUT AccessToken, Out RenewalFreq,
                 OUT AppServerSet)
ReleaseAccessToken(IN AccessToken, IN UserName, IN Password,
                   OUT ReturnCodes)
RefreshAppServerSet(IN AccessToken, IN BadQos, IN NoService,
                    OUT ReturnCodes, OUT ServerSet)
```

LSM Functionality Overview

The web pages set up at \\fserv2\home\webpages\pmain.html are intended to provide a framework for discussing overall issues related to subscribe/install/uninstall scenarios, and I look forward to reviewing them with everyone, soliciting feedback, & updating the pages & requirements document appropriately. The material in this section is presented at a much lower level than the web pages.

Run an Application

- ECM invokes **LSMNotifyAppStart** whenever there is a transition from 0 to 1 open files wrt a particular eStream app. ECM can determine the AppID of an

eStream app by inspecting the fully specified eFSD filename; the first level of the directory path is a string version of the AppID. If LSM does not recognize this AppID for some reason, an appropriate status is returned to ECM.

- LSM sends an **AcquireAccessToken** call to via CNI to a SLiMServer [possibly only if the ExpectedExpirationTime for the AppID in question is in the past] and when LSM receives the AccessToken, it places a copy of it in the Subscription Data Set, assigning ExpectedExpirationTime according to the RenewalFrequency indicated by the SLiMServer and setting up a timer alarm to go off shortly before the AccessToken is expected to expire. LSM marks the AppID as ActiveInUse in the Subscription Data Set and returns an appropriate status to ECM.
- Should an AccessToken be denied due to a mandatory upgrade, LSM retries AcquireAccessToken w/ the new RootFileNumber, updates the RootFileNumber in the associated Subscription Data Set entry, & calls **InformUser("Upgrade")**. The RootFileNumber for the specified AppID is always returned to the ECM as an OUT paramater of **LSMNotifyAppStart** so that ECM can take appropriate action if the application version has changed from that being cached.
- Should an AccessToken be denied because it is currently held by another of the user's clients, a copy of the other client's AccessToken is returned. If the user responds affirmatively to **AskUserYesNo("Yank token?")**, then LSM will call **InformUser("Please wait")**, send the token to **ReleaseAccessToken**, & then retry **AcquireAccessToken** at the appropriate time.
- CNI invokes **LSMGetSLiMServerSet** to obtain SliMServers for AccessToken requests. Should CNI have quality of service issues with those servers, it calls **LSMGetNewSLiMServerSet**; LSM uses the **PromptASPNotify** interface to engender the ASP Web Server to send a message containing a SliMServerSet.
- Whenever an AccessToken timer alarm goes off, LSM checks that the associated application is marked ActiveInUse and if so, it issues a SLiMServer request to **RenewAccessToken** and records the returned token in the Subscription Data Set, resetting ExpectedExpirationTime and scheduling another timer alarm.
- ECM invokes **LSMCheckAccessToken** whenever it receives an eFSD request (whether the associated data is currently in ECM's cache or not). If LSM does not have an AccessToken or if ExpectedExpirationTime is in the past, LSM invokes **AcquireAccessToken**. If ECM needs to fetch the requested data from an AppServer, it invokes **LSMGetAccessToken**.
- CNI frequently calls **LSMGetAppServerSet** to obtain a set of AppServers through which to cycle; LSM returns the servers associated with the most recently acquired AccessToken. Should CNI be dissatisfied with the quality of service associated with the AppServers it is given, it calls **LSMGetNewAppServerSet** which sends **RefreshAppServerSet** to a SLiMServer.
- Should an CNI request be denied by an AppServer because the submitted AccessToken was expired, CNI returns an appropriate status to its caller. If the caller was attempting to satisfy an access from an ActiveInUse application, it will undoubtably call LSMGetAccessToken.
- ECM invokes **LSMNotifyAppStop** whenever there is a transition from 1 to 0 open files wrt a particular eStream app. LSM sends a **ReleaseAccessToken** call to a SLiMServer [possibly only if the ExpectedExpirationTime for the AppID in

question is not in the past]. LSM marks the application as Inactive in the Subscription Data Set, which indicates that automatic AccessToken renewal should not be continued for this application [LSM may cancel outstanding alarm].

Get/Refresh Subscription Information

- Whenever the CBM receives an **ASPNotify** from an ASPWebServer, it sends an **LSMAspAccountPing** message to the LSM. The LSM checks its ASP Data Set to see if the specified ASP is one that it recognizes. If LSM does not recognize the ASP, it allocates a new ASP Data Set entry and calls **AskUserPassword** to prompt the user for the password associated with this ASP+UserName (asking the user if the password can be saved on the client for future use). If LSM does recognize the ASP, it only calls **AskUserPassword** if the user did not allow the password to be previously recorded.
- Once LSM has the UserName & Password, it issues a **GetSubscriptionList** request to one of the ASP's SLiMServers. For any SubscriptionIDs associated with that ASP that are installed on the client but do not appear in the returned list, the user is informed that he cannot run the app at the current time & is asked if he would like it uninstalled from his client (note that we may be legally required to remove it from the cache, so this may not be a question but rather just a message); **AIMUninstall** is invoked to achieve this.
- For all other SubscriptionIDs, a **GetLatestApplicationInfo** request is sent to a SLiMServer.
- For SubscriptionIDs already in the Subscription Data Set, the RootFileNumber is compared to see if the application has been upgraded. If it has, the user is informed; the new RootFileNumber is recorded for the next run (assuming either that the upgrade is mandatory or that the user indicated he wanted it).
- For SubscriptionIDs that were not already in the Subscription Data Set, a new Subscription Data Set entry is created and **AskUserYesNo("Install App?")** is invoked. If user responds in the affirmative, LSM calls **ECMMount** to notify ECM to prepare to service requests on behalf of this new application & then invokes **AIMInstall,** which can use the eFSD/ECM path to obtain the AppInstallBlock. If user responds in the negative, he is able to indicate "don't ask again" or "ask again later", which is recorded in the Subscription Data Set.
- Whenever the eStream client software starts up or at specific times requested by the user via the ClientUI, **LSMUpdateAllSubscriptionStatus** is called to refresh the eStream client's knowledge of the apps to which the user is subscribed.

Display Subscription Information

- The CUI provides a user interface to display ASP and application info, using **LSMGetAspList, LSMGetAspInfo, LSMGetAppList, LSMGetAppInfo.**

Client Browser Module Overview

The CBM is an eStream client component that runs in association with the client browser. A must-have item is that it be able to communicate a small amount of data in some way to the core eStream client software. A strongly-desirable item is that it be realizable on both Internet Explorer 4+ and Netscape Navigator 4+. A would-be-nice item is that it be able to start the eStream client software if it were not already running.

A number of technologies are currently under investigation. I believe that all three of the items above can be met by native code distributed as a browser plugin, which registers to handle a particular MIME encoded filetype (*.estream) in the HTML stream. Bhaven has created JavaScript that can start Excel & communicate with it, though my understanding of this solution is that it required that the browser be Internet Explorer, that the app be ActiveX, and that the security setting on the browser be at minimum setting.

Issues

- Decide how to store ASP Data Set and Subscription Data Set in nonvolatile memory on the client system.
- Develop mechanism for implementing client browser module.
- Refine needs with respect to Client User Interface.
- Add Prefetcher interface.

eStream App Server Low Level Design

Version 1.2

Sameer Panwar
[redacted]

Functionality

First, some definitions:

eStream page: the smallest unit of data that can be requested by a client from an App Server. Proposed to be 4kB for eStream 1.0.

page set: simply, a sorted list of eStream pages, each identified by a File ID (i.e. AppID & File #) and page # (essentially an offset into the file). This set is restricted only in that all pages in the set must have the same AppID.

client request: a single self-contained message from a client requesting a page set from the server. Each server response to a client request can return a number of pages, and there is a maximum number of pages that the client can request in this message. (TBD, somewhere between 8 and 20 or so).

The primary job of the App Server is to service client requests for application data blocks. The App Server is designed to minimize the amount of CPU time it must consume to satisfy each client request, thereby maximizing scalability. Thus, authentication is performed by a simple expiration time check of an AccessToken provided by the client, and compressed application data is saved persistently.

The App Server serves data derived from eStream Sets. To decouple the performance needs of the App Server from the Builder, we should have a post-processing tool that converts the flat, uncompressed eStream Sets as provided by the Builder into a precompressed format suitable for memory mapping, if the App Server is configured to serve compressed bits. Also, a profiling part of the App Server can be used to monitor for common page sets, and then assemble more optimized replies, which compress the set of pages together as a unit, to take advantage of improved compression ratios. These replies can be stored on disk to save time in rebuilding them each time the server is started up.

The App Server (AS henceforth) views an eStream Set as simply a set of files, and knows no further underlying structure. Thus an eStream Set contains at the start a table (FOST) indexed by File #, and providing the offset into the eStream Set where the associated file data begins, and the size of the file. So the AS just takes the client request of (AppID, File #, Page #, no. of pages), maps AppID to an eStream Set and looks up in the FOST table (File/Offset/Size Table) to find the requested data.

This works slightly differently when the eStream Set file has been pre-compressed by the post-processing tool. The resulting image is the same as before, except now the FOST points to another table, the POST (Page/Offset/Size Table). Because the compressed pages will be of different sizes, this table must by indexed by the Page # to find the relative offset and size of the compressed page data for the file. Thus if an AS is not configured for data compression, the main difference in behavior is that it doesn't do a POST lookup and it doesn't care about coalescing page sequences.

Data type & Data structure definitions

Processed eStream set – this structure is kept on disk and never changes after installation. It looks like:

```
struct {
      ApplicationID appID;      /* for reference, is a 128-bit GUID, see ECM
LLD */
      uint32 maxFileNo;
      boolean compressed_flag;    /* indicates whether the AppFiles are compressed, though maybe we should do it differently? */
      FOST_Entry FOST[<maxFileNo>];
      uint8 appData[<sum of all AppFile sizes, which are variable>];
} ProcessedEstreamSet;
```

Since the files in the application are of variable size, we can't make a table out of them, and must indirect out of a table (indexed by the File #) to find their offset location inside the AppData buffer.

```
struct {
      uint32 offset;
      uint32 size;
} FOST_Entry;
```

When the processed eStream set is compressed, then we use the AppFileCompressed structure at the offset indicated by the FOST, otherwise we interpret the data as just AppFile. The AppFileCompressed structure starts with a table that indicates the size and offset of the compressed data that belong to the page it was indexed by.

```
struct {
      uint8 fileData[<size from FOST_entry>]
} AppFile;

struct {
      POST_Entry POST[<number of pages, derived from size from FOST_Entry>];
      uint8 fileData[<sum of all FilePage sizes, which are variable>];
} AppFileCompressed;

struct {
      uint32 offset;
```

```
    uint32 size;
} POST_Entry;
```

This covers all the structures that live on disk. When we mmap-per-file, that means we make multiple mappings out of a single ProcessedEstreamSet file, at different offsets, one for each file.

Now, for the in-memory data structures (assuming per-file-mmapping):

The primary lookup will be a hash table, hashed on the AppID and FileNo. It should have on the order of 10,000 entries, each table entry containing a list of entries (for collisions). Each list entry contains:

```
struct {
    ApplicationID appID;
    uint32 fileNo;
    uint32 size;   /* size of the mapped Appfile */
    MMap fileMap;
    HTListEntry * next;
} HTListEntry;
```

The Mmap struct just contains any OS-specific-related stuff to manage the mappings, plus a field `char * ptr`, which points to the place in memory that the AppFile (or AppFileCompressed) is mapped. So the hash table looks like:

```
struct {
    HTListEntry * entry[<size of hash table>];
} MMapHT;
```

Hash function is TBD. The hash table should be statically sized large enough to handle the full number of eStream sets up to the maximum memory we will support. Assuming 32 bytes being used per entry, that implies about 1 MB to handle 30k files, which is no problem. (Maybe we should reserve entries for 100k files or more?)

Configuration: Each AS must obtain configuration data, either directly from the database or from the monitor in its startup message. The required data is (with the config param names and datatypes):

```
AppList          vector of ApplicationID's (128-bit GUIDs)
ServerPort       uint16
MonitorPort      uint16
SLiMKey          uint (size TBD, depends on actual algorithm)
ClientTimeOut    uint32
CompressionFlag  uint32
```

Network communication: The AS talks only to clients and the server monitor via the network. The server monitor communication will be described as part of the monitor heartbeat protocol. The AS-client communication will be described in a separate docu-

ment. The AS will time-out and close connections that have been idle for some amount of time (a few seconds).

[maybe combine multiple responses into a single send socket call (will only work for TCP probably, since proxies won't like multiple server responses)?]

Interface definitions

The AS is optimized to do one thing only: serve pages from the read-only file system part of eStream, so there is just one interface with the client. Anything the client can care about in an eStream set is just another file to the AS, including the AppInstallBlock, and directories/metadata. The AS only returns the data the client requested, nothing extra.

```
struct {
     uint32 fileNo;
     uint32 pageNo;
} PageRequest;

struct {
     uint32 errorCode;
     uint32 compressedFlag;
     uint32 fileNo;
     uint32 pageNo;
     uint32 offset; /* offset into pageData below */
     uint32 dataSize;
} PageReply;
```

PageReadRequest

Caller:	Client	
Callee:	AppServer	
Input:	uint32	appId;
	eStreamAccessToken	accessToken;
	uint32	numPagesRequested;
	PageRequest	pageSet[(numPagesRequested)];
Output:	uint32	numPagesRequested;
	PageReply	pageSetReply[(numPagesRequested)];
	uint8	pageData[(sum of all page data)];
	uint32	globalErrorCode;
Global Errors:	INVALID_ACCESS_TOKEN	
	EXPIRED_ACCESS_TOKEN	

INVALID_APP_ID
EXCEEDED_MAX_REQUESTABLE_PAGES

Errors within
PageReply: INVALID_FILE_NO
INVALID_PAGE_NO
SERVER_ERROR (probably should be logged, and should cause an alert if too many occur in some time period, including errors that don't get returned to the client.)

AppServers don't ever talk to the database (it would be a waste of licenses considering the number of AppServers we'd have and their infrequent accesses). Instead, they obtain all their relevant control information from the server monitor.

The exact interfaces are TBD, but from the monitor they will provide configuration information, AppServer state change requests, and add/remove requests to the list of apps being served. Going back from the AppServer to the monitor, it will report load (average response time) on a per app basis, and server state, along with the heartbeat.

Component design

Interesting issues to deal with:

Scalability/Performance

Since scalability (and thus performance) is critical for the AS, let's go over how CPU and memory are used.

Memory

Performance is maximized when virtually all client requests can be satisfied by retrieving the desired pages from RAM, because RAM is far faster than disk. Thus the amount of RAM available will put an upper bound on the number of apps that a single AS can serve efficiently. Since server RAM won't grow as fast as the total size of all apps available as eStream sets, this means we'll have to heterogenize servers, where each server specializes in a subset of apps, limited by available RAM. For eStream 1.0, this component of AS configuration will be handled manually, the eStream administrator assigning apps to servers. In the future, the set of App Servers should automatically reassign apps dynamically to balance load.

But this is just one level of memory, committing RAM to a set of apps. There still remains the question of how to best utilize that RAM for each app, since some files are used far more often than others. This immediately means that for efficiency we must overcommit RAM, because if we allocate an entire eStream set into RAM, we're using precious resource to hold data that may be requested only very rarely. Instead of having to manage our physical RAM manually to accomplish this (such as with a cache), an easier approach would be to take advantage of virtual memory (VM) to automatically keep

the hot pages in RAM, with the remainder available (again **automatically**) off disk (via memory mapping the eStream sets). That way the server can satisfy any possible client request for any app it serves, but is optimized to be the most efficient over all clients. But this only works if enough VM is available. (Time for some back-of-the-envelope numbers.) Given that an app seems to have something like only 20% of it being hot (from our current limited data from the prototype), this means VM must be at least 5x of RAM for maximum efficiency. Given that a process has about 2 GB addressable VM, this corresponds to about 400 MB of RAM. Beyond that size (which is not uncommon), we don't have enough VM to efficiently overcommit our memory (by mmaping entire eStream sets). So now our choice is to either manually manage a memory cache (and all the attendant coding, bugs, etc.), or to mmap at a finer granularity.

Note that the effective virtual memory required by an app is increased when compression is used, to handle the extra compressed page sets. They'll probably double or triple the RAM footprint by hot pages (due to redundancy), but only increase the overall VM footprint by 1.2 – 1.5. The consequence of this is that the overcommit ratio goes down to 1.5 / (3 * .2) = 2.5, though the amount of apps servable is reduced to 1/3 (!!). Now 2 GB virtual address space corresponds to 800 MB of RAM. This means we should be able to just memory map entire eStream sets, up to 2 GB worth, and be confident we're utilizing RAM efficiently, assuming the server has about 800 MB of RAM. A server with less RAM will likely thrash, and those with more will likely see little improvement in the number of apps they can serve via memory mapping.

A loss of 2/3 in the number of apps an AS can serve I think is too great a sacrifice, too great a loss in app scalability (need 3x the number of servers as before!) for what is about a 15-30% greater effective bandwidth at the client. The root of this problem is the redundancy (costly in physical memory), because the compressed page sets will contain the same page in multiple sets. This is similar to the redundancy that appears in trace processors and dynamic translation, which places extra memory demands in both those cases. I think we must completely eliminate this redundancy to achieve the goals we desire, either by (1) not using compressed page sets, and just sending multiple individually compressed pages, or (2) ensuring a page appears in only one compressed page set. [There further potential loss of effective memory size when using compressed page sets since they'll be allocated in 4k chunks, thus wasting about 2k on average; we'd have to batch them up together in files to minimize this... Also, saving the compressed page sets to disk introduces extra complexity to the AS because we'd have to properly handle recovery (i.e. what if the system crashes while we're writing the sets, which if we're memory mapping is totally out of our control). Because of this robustness requirement, and the fact we need to be 100% sure we're serving good bits (lest we crash a bunch of clients), this needs to be thought out very carefully if we want to do this. My opinion is that we should defer implementing compressed page sets until we better understand the tradeoffs, and good profiling schemes. In particular, will the AS be mostly bandwidth-limited, memory-limited or CPU-limited?]

Separately from this, we should consider the effect of per-file memory mapping (ignore the compressed page sets now). This has the impact of requiring many more mmap's

from the OS, but promises better use of the limited virtual address space. In this scheme, we mmap each file into VM as it is referenced by a client. If only hot files are referenced, then the RAM footprint is the same as before, but VM is only used for the hot files, not the entire app, probably about 30-50% greater in size. Thus the overcommit ratio then becomes 1.5, much better than the 5 with full app mmapping. So 2 GB of VM corresponds to 1.3 GB of RAM, much better than the 400 MB with full app mmapping. However, this assumes that VM is used in a cache-like manner, evicting not recently used mmap's, since as uncommon files are referenced, they eat up more and more VM. Once VM is totally used up, then replacement policies and eviction (and fragmentation of virtual address space) become issues, just as with a manually managed cache. One solution is to simply purge all mmap's and start from scratch, which is simple and reliable, especially considering the AS is multithreaded (if this is done, the above analysis doesn't hold, and performance becomes a function of how often VM is cleared). Another possibility might be to use the profiling mechanism and only place sufficiently popular files in mmaps and do regular file system accesses for the rest.

Of course, the alternate option for managing physical memory is to know its size, and manage a cache manually. One advantage here is that the AS would know the physical memory consumption and usage (unlike when the OS was handling everything), which may help with load balancing. The main advantage is that there are no artificial limits (overcommit ratio is irrelevant), and only physical memory size is the true limit, and this approach can map any number of eStream sets (with any size of files) to any amount of physical memory up to the virtual address size (4 GB). Then memory management becomes an issue (what do you do once all your RAM is full), which can be painful in a multithreaded environment. Again, we can just invalidate the whole cache as an option, but this will probably happen more often than with the per-file-mmapping case, unless RAM is greater than the maximum that the per-file-mmaping approach can handle. If the wholesale cleanup approach is used, then allocating fixed size chunks may not be needed, and we could potentially get better memory usage by packing compressed pages more tightly (e.g. 16-byte aligned vs. 4kB aligned), which is another potential advantage. Maybe instead of wholesale cleanup, we mark the most commonly used pages, and then just compact those and dump the rest (say 50%). The main issue with this approach is potential redundancy with respect to the OS disk cache (which is shared in the mmap approach), and assumption that our caching policies will be better than the OS's. Also, lookups get messier, since we need a bigger lookup table to index via page # as well.

Yet another option is to use multiple processes instead of multiple threads, one process per app being served, thereby releasing us from the 2 GB VM limitation. However, this introduces the issue of multiplexing requests from the network via IPC, and more load on the server monitor. On x86 NT, a Very Large Memory feature is available that can provide 36-bit addressing per process; we may want to use this even though it won't be available on regular Unixes (and probably not x86-linux).

In summary: per-eStream-set-mmapping is probably too wasteful of virtual address space. Per-file-mmapping is much better, but then memory management becomes an issue, suggesting a simple throw-away-and-start-over solution. However, given that solu-